ESD-TR-75-61

MTR-2922

PARALLEL DERIVATIVES FOR MULTIPROCESSOR TASK SCHEDULING

J. K. Millen

MAY 1975

Prepared for

DEPUTY FOR DEVELOPMENT PLANS
ELECTRONIC SYSTEMS DIVISION
AIR FORCE SYSTEMS COMMAND
UNITED STATES AIR FORCE
Hanscom Air Force Base, Bedford, Massachusetts

Project No. 7090
Prepared by
THE MITRE CORPORATION
Bedford, Massachusetts
Contract No. F19628-75-C-0001

REVIEW AND APPROVAL

This technical report has been reviewed and is approved for publication.

Jack Segal

JACK SEGAL, GS-13
Project Officer, Project 7090

[illegible]

J. L. MASI, Colonel, USAF
Director, OASIS Program Office

FOR THE COMMANDER

[illegible]

THORNTON T. DOSS, Colonel, USAF
Deputy for Development Plans

UNCLASSIFIED
SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER ESD-TR-75-61	2. GOVT ACCESSION NO.	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) PARALLEL DERIVATIVES FOR MULTIPROCESSOR TASK SCHEDULING		5. TYPE OF REPORT & PERIOD COVERED
		6. PERFORMING ORG. REPORT NUMBER MTR-2922
7. AUTHOR(s) J. K. Millen		8. CONTRACT OR GRANT NUMBER(s) F19628-75-C-0001
9. PERFORMING ORGANIZATION NAME AND ADDRESS The MITRE Corporation Box 208 Bedford, MA, 01730		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS Project No. 7090
11. CONTROLLING OFFICE NAME AND ADDRESS Deputy for Development Plans Electron Systems Division, AFSC Hanscom Air Force Base, Bedford, MA, 01731		12. REPORT DATE MAY 1975
		13. NUMBER OF PAGES 95
14. MONITORING AGENCY NAME & ADDRESS(if different from Controlling Office)		15. SECURITY CLASS. (of this report) UNCLASSIFIED
		15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

Approved for public release; distribution unlimited.

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

MULTIPROCESSING
PARALLEL PROCESSING
TASK SCHEDULING

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

Three alternatives are presented for the design of a scheduler for a multi-miniprocessor system. The alternatives are based in general on the theory of parallel program schemata and in particular on a new parallel derivative technique for constructing the maximally parallel flowchart equivalent to a given one.

TABLE OF CONTENTS

TABLE OF CONTENTS (CONCLUDED)

LIST OF ILLUSTRATIONS

LIST OF TABLES

LIST OF ILLUSTRATIONS (CONCLUDED)

LIST OF PROGRAMS

SECTION I

INTRODUCTION

MULTIPROCESSOR SCHEDULING

This report is concerned with asynchronous or multiple-instruction-stream computers in which processors may operate independently except for the intercommunication necessary to cooperate in the execution of a single program.

The ease of keeping a large number of processors busy depends to some extent on the makeup of the population of jobs awaiting execution. If most of the jobs are independent from each other as far as order of execution is concerned, jobs can be introduced by a multiprogramming scheduler as soon as the resources they will need become available. Greater utilization of processors can be obtained by scheduling some jobs when enough resources to get them started become available; care must be taken to avoid deadlock in such schedulers.

PARALLEL ANALYSIS

If the population of jobs is small, or the scheduler's choice is restricted by ordering constraints among jobs, a higher utilization of processors can often be achieved by decomposing the existing jobs into tasks. While the tasks will generally have many ordering constraints among themselves, it is usually possible to divide a job into tasks in such a way that two or more tasks can at times be executing concurrently.

Division of a job into tasks and specifying which of them may be executed in parallel with others can be done by the programmer or, even after the program has been written serially, by parallel analysis techniques.

SOURCE LANGUAGES WITH PARALLELISM

Programmers can be given control over parallelism by providing them with instructions like FORK and JOIN and their extensions like DO SIM (-ultaneously) and DO CONC (-urrently). Array instructions such as those in APL also imply parallel execution in a natural way. At a higher level, intercommunication between subroutines might be formulated in a way which encourages parallelism, such as by using semaphores for synchronization of conceptually independent routines which share data areas, or limiting communication to producer-consumer relationships. Simulation languages provide examples of these concepts. More sophisticated examples can be found in the literature on Petri nets and related notations.

KUCK'S RESULTS

Parallel analysis of programs can uncover additional opportunities for concurrent execution. The amount of potential parallelism in ordinary serial programs is surprisingly high, according to the results of a study [1] by D. Kuck, et al, at the University of Illinois. They analyzed 86 FORTRAN programs representing a variety of applications, and estimated the possible speedup (serial execution time/parallel execution time) for each program, and the number of processors needed to achieve that maximum speedup. Tasks were statements, or, in the case of arithmetic expressions, sub-statement operations. Figure 1 shows their results. It has a point for each of seven classes of similar programs for which the results were averaged, and the overall average was a speedup of 10 with 35 processors. These results do not take account of scheduler overhead, and therefore will be degraded in practice; on the other hand, the parallel analysis methods were described by the authors as "crude in several respects", so the speedups, if not the utilization, might be improved.

RESEARCH IN PARALLEL ANALYSIS

Parallel analysis is still a research topic. A review of some of the past work is given by Gonzalez and Ramamoorthy [2], along with some original methods. Some new algorithms were developed by Kuck, et al, for their analysis [3]. Techniques for handling nested DO-

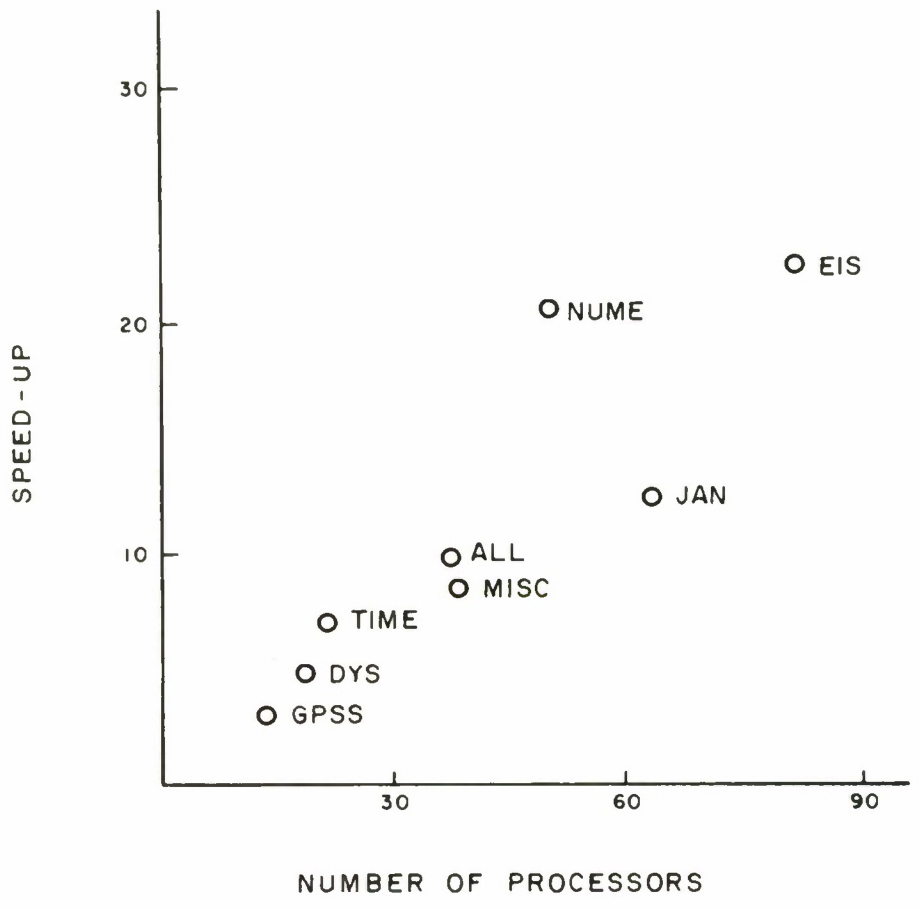


Figure 1 KUCK'S RESULTS

loops have been discovered by L. Lamport [4]. (Although one of these, the "coordinate method," is intended for single-instruction-stream computers, his related "hyperplane method" applies to multi-processors.)

The work cited above on parallel analysis was done before an important paper by R. M. Keller [5] appeared, giving fundamental results on maximal parallelism. Keller's results will probably have their greatest impact on global program analysis, the area in which the above techniques are weakest. Keller's theoretical framework discouraged him from considering certain problems and opportunities involving DO-loops, arithmetic expressions, and other topics, but it appears to be flexible enough so that appropriate extensions are worth trying.

PARALLEL DERIVATIVE

Keller's paper includes an algorithm for parallel analysis. It was exhibited primarily for the purpose of demonstrating its existence, and was not intended to be computationally efficient. In order to retain the theoretical advantages and clarity of Keller's results, in a practical algorithm suitable for computer implementation, the parallel derivative was introduced in this project.

The parallel derivative is a tool for determining which tasks in a flowchart can be initiated in parallel at a given point in the computation. In its mathematical form, it is an operator on sets of sequences representing the computations indicated by a flowchart. If the flowchart is implemented as a list structure or character string, the parallel derivative can be implemented as a recursive procedure which searches the flowchart to an extent determined by the ordering constraints among the tasks.

The principal advantage of using the parallel derivative (aside from convenience of implementation), or other techniques based on parallel program schemata, is the guarantee of maximal parallelism permitted by the theory.

The principal shortcoming is the fact that there are many ways to transform a source program into the flowchart form, especially because of DO-loops and options in the treatment of temporary variables. A poor reduction to flowchart form can boil away potential parallelism.

SUMMARY OF THIS REPORT

This report begins with an introduction to the theory of parallel program schemata, with an emphasis on flowcharts, both serial and parallel. Some of the subtleties arising in the transformation of a program to a suitable flowchart are introduced, but not investigated systematically. Others, such as the repetition-free condition, are not significant in practical situations and are not mentioned in the exposition.

A working definition of the parallel derivative and a method for using it to generate the maximally parallel equivalent flowchart, the closure, are presented. Because of the size of the closure, condensed formats for it are of interest. One such format is the queue realization, whose size is often not much larger than the original flowchart, although in many cases it falls somewhat short of maximal parallelism. The best queue realization is based on a serial flowchart equivalent to the given one, obtained from it by a "pulling" transformation which can be implemented with the help of the parallel derivative.

A multiprocessing scheduler can be constructed from the closure in three ways, whose advantages and disadvantages arise from the tradeoffs made in size vs. speed, and preprocessing vs. run-time processing.

The three types of schedulers can be experimented with on a multiprocessor in which each processor is inexpensive. Modules manufactured by Intel Corporation for their Intellec-8 microprocessor development system were found to be adequate for configuring a multiprocessor with a potentially unlimited number of processors. Considerations for implementing the intercommunication between these processors are discussed in Section VII.

SECTION II

PARALLEL FLOWCHARTS

FLOWCHARTS AND OPERATIONS

A parallel program schema is a representation of a class of similiar programs. Its elements are:

1. a set of operations,
2. the flow of control between operations.

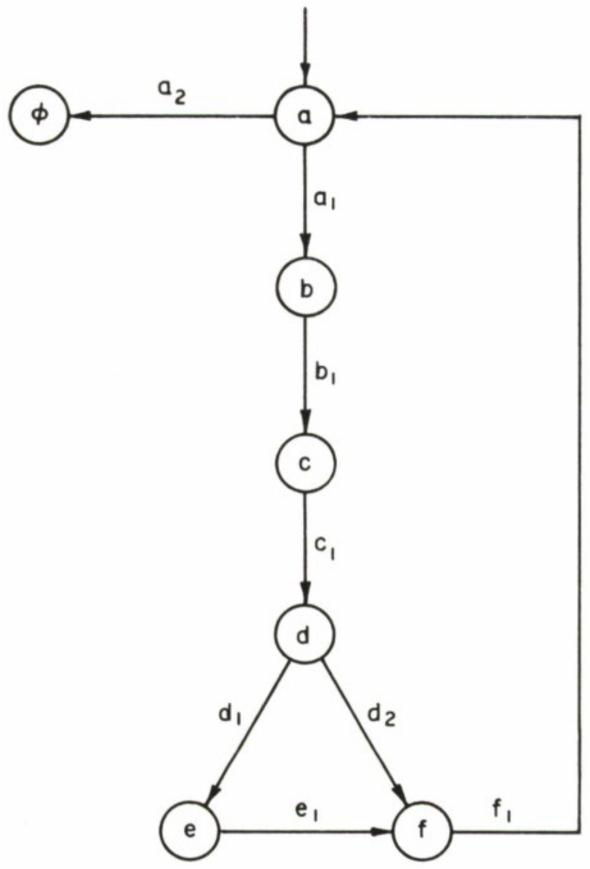
An operation is specified by:

1. the set of memory locations it reads,
2. the set of memory locations it writes into,
3. the number of outcomes it has.

An operation may represent a subroutine, a sequence of source language statements, a single statement, a machine language instruction in the expansion of a source language statement, or any part of the program which can be described by the three elements above. In our application, an operation will most likely be either a single source language statement or a short sequence of statements. The outcomes of an operation are distinguished only in so far as they affect the subsequent flow of control.

ENABLED OPERATIONS IN STATES

The flow of control is shown by a flowchart, like the one in Figure 2. The circles are *states*. Symbols written in the circles are operations which may be executed when the state is entered; they are said to be *enabled* in that state. The empty set symbol $\emptyset$, occupies a



1A-44,387

Figure 2 A FLOWCHART

final state in which no operations are enabled.

TRANSITIONS AND TERMINATORS

Arrows from one state to another are transitions. The arrow with no state at its tail marks the initial state at its head. Each transition is labeled with a symbol called a terminator which represents an outcome of one of the operations enabled in the state from which the arrow departs. By convention, the terminators of an operation b are $b_1, b_2, \ldots$; there are as many terminators of an operation as outcomes, and every outcome determines a transition.

The flowchart in Figure 2, together with the operation specifications in Table I, defines a particular program schema. It represents any program which can be divided into operations satisfying Table I and then flowcharted as in Figure 2. An example of such a program is in Table II.

Table I

Operation Specifications Example

OPERATION	NUMBER OF OUTCOMES	WRITES	READS
a	2	X,Y	Y
b	1	Z	X
c	1	Z	X
d	2		X
e	1	X	X
f	1	Z	X

Table II

A Program and Its Operations

PROGRAM		OPERATIONS	OUTCOMES
L1:	READ X [ON EOF RETURN]	a: $X \leftarrow Y \leftarrow ?$	a_1; a_2 = End of File
	PRINT X	b: $Z \leftarrow X$	b_1
	PRINT X^2	c: $Z \leftarrow X^2$	c_1
	IF $X \geq 0$ THEN GO TO L2	d: compare X to 0	$d_1 = <$; $d_2 = \geq$
	$X \leftarrow - X$	e: $X \leftarrow - X$	e_1
L2:	PRINT $\sqrt{X}$	f: $Z \leftarrow \sqrt{X}$	f_1
	GO TO L1		

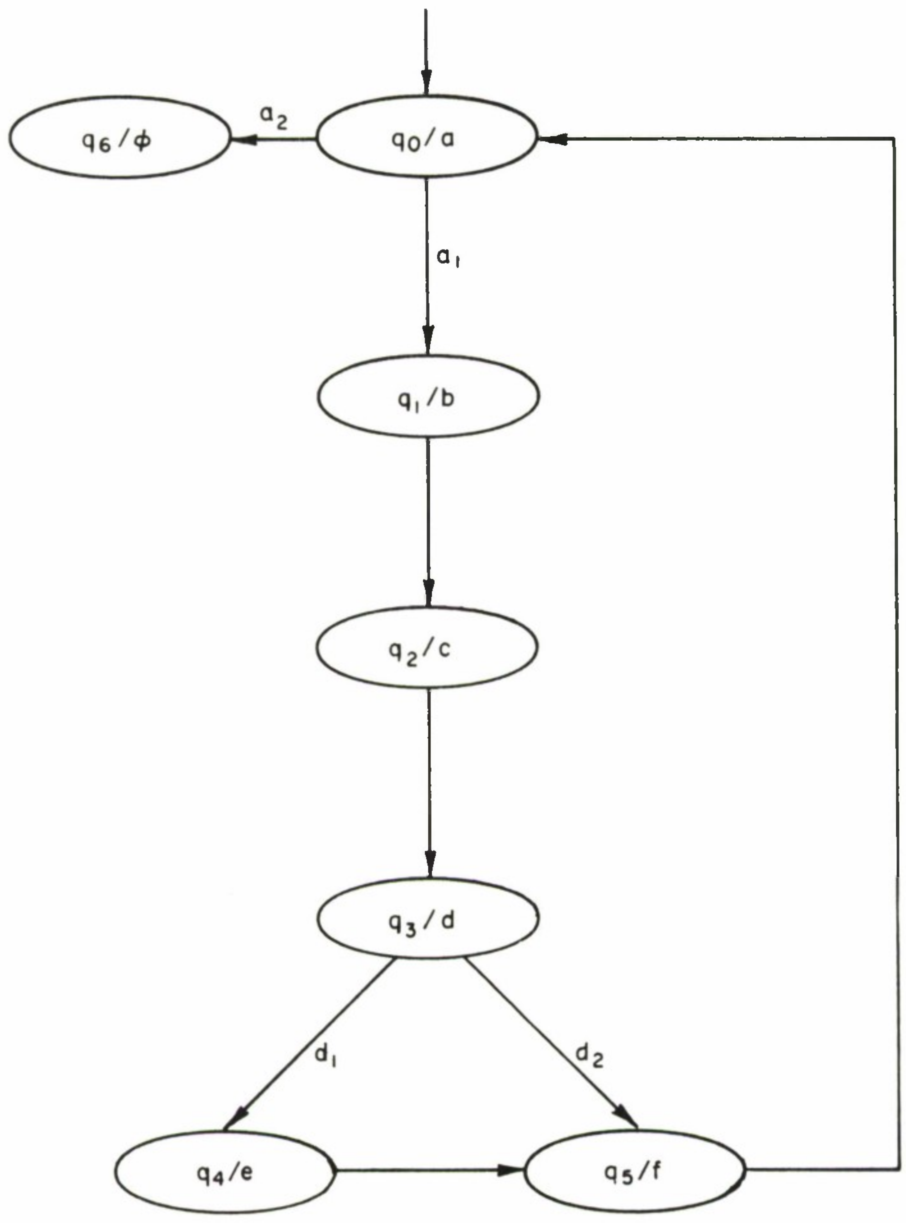
In the example, each statement has been taken as an operation, except "GO TO" statements. A print statement is considered as a write to an "output" memory location. A read statement is a read of an "input" memory location, but it is a write as well because it also has the effect of filling the input first with the next record. Note that b, c, and f are different statements, because they read different variables.

SERIAL FLOWCHARTS AND EQUIVALENCE

The flowchart in Figure 2 is _serial_ because at most one operation is enabled in each state. In order to introduce parallelism into the program, we will add more operations to some states and create new states as well. The result will be a different parallel program schema, but it is still considered _equivalent_ to the serial one we started with, because when the program is executed in parallel under the control of the new flowchart, _each memory location will go through the same sequence of values that it did before_. This notion of equivalence is strong enough to preserve the external behavior of the program. In fact, it is stronger than necessary, because parallelism can often be increased by creating new temporary variables whose use violates equivalence but clears a bottleneck.

ENABLING OPERATIONS EARLIER

It will be convenient to refer to a state by a name independent of the operations enabled in it. We will use symbols of the form q_i,



IA-44,399

Figure 3 THE FLOWCHART IN FIGURE 2 WITH STATE NAMES $q_0, \cdots, q_6$ ADDED AND SOME LABELS OMITTED

where i is an integer, for this purpose. The flowchart of the example has been reproduced in Figure 3 with state names added, and transition labels omitted where they are redundant.

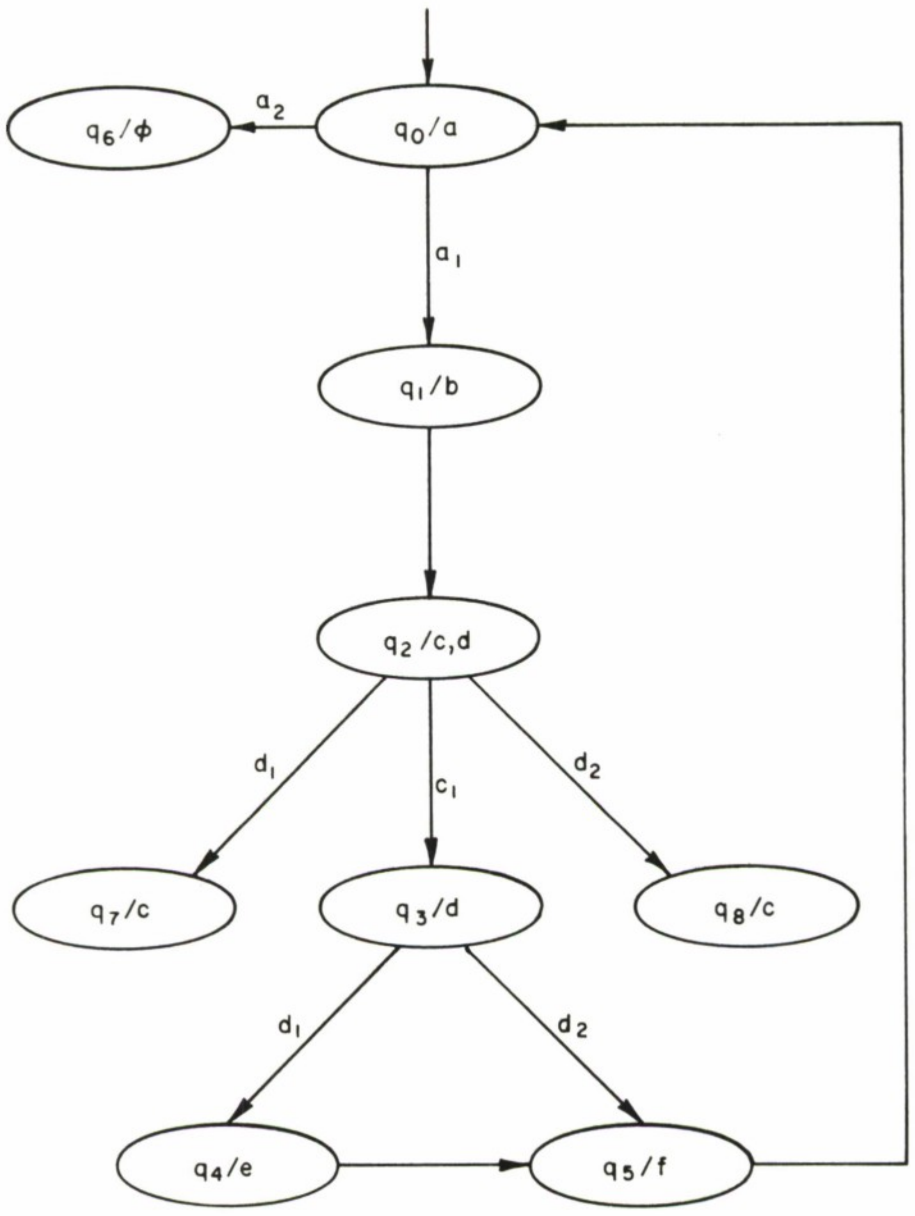
Intuitively, we can see an opportunity for parallel execution in state q_2. We would enable d in q_2 along with c, because the only variable written into by the two operations, Z, will receive the same value even if the order of execution of c and d is reversed. In this case we say that c and d <u>commute</u>. Two operations will always commute if all variables referenced by both of them are only read by both, but not written by either. The memory conflict for reading a common variable, like X in the example, is not a concern of this theory.

The commutativity relation for the example, based on the above criterion, is shown in Table III.

Table III

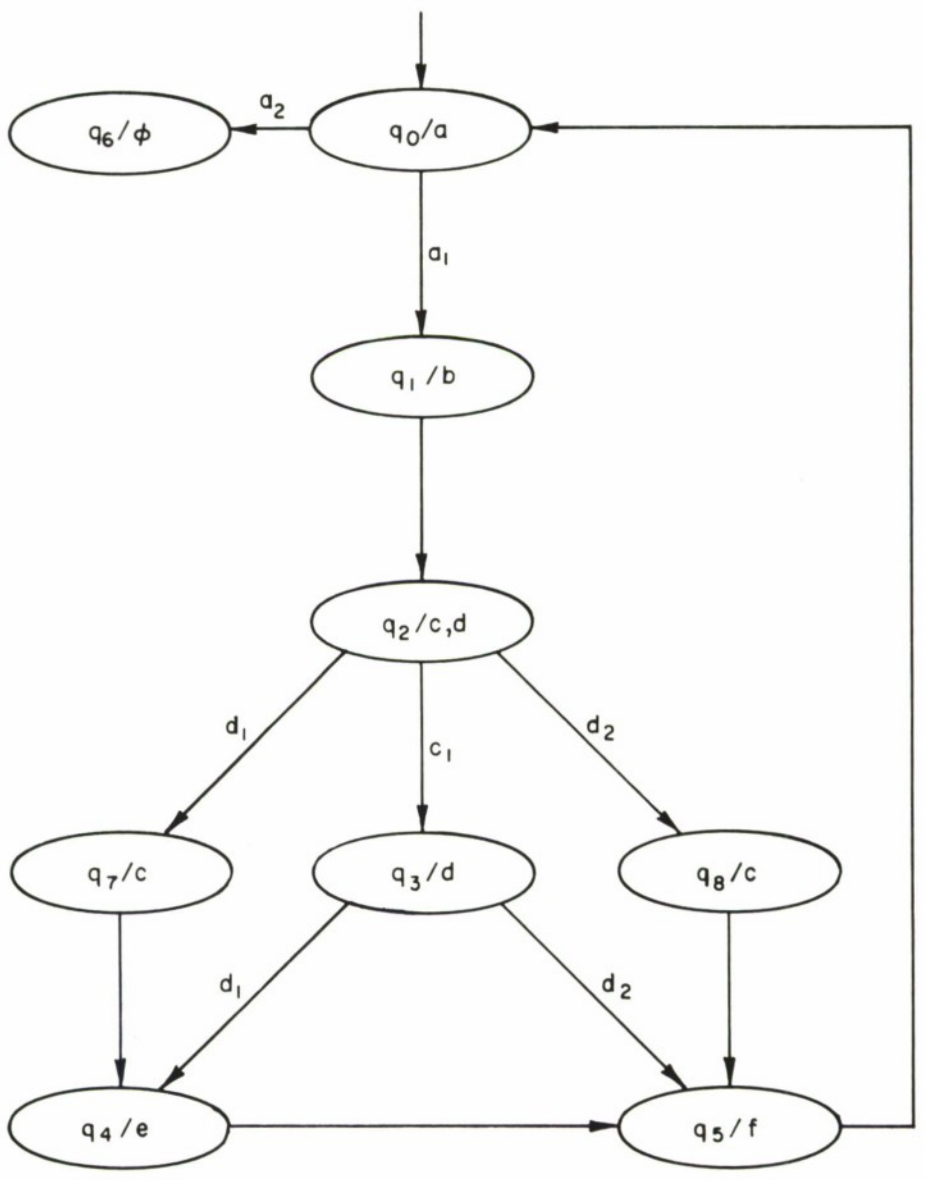
Commutativity Relation for the Operations in Table I

OPERATION	COMMUTES WITH
a	none
b	d
c	d
d	b, c, f
e	none
f	d



IA-44,398

Figure 4 OPERATION d ENABLED IN q_2 IN ADDITION TO c



IA-44,397

Figure 5 A PARALLEL FLOWCHART EQUIVALENT TO THE FLOWCHART IN FIGURE 2

NEW STATES: PERSISTENCE

If d is enabled in q_1 as in Figure 4, transitions associated with its termination go from q_2 to newly created states, say q_7 and q_8, as shown. In q_7 and q_8, operation d has been completed, but c is still enabled. This is because if d finished first, then c either is still executing or has not yet had a chance to be initiated. In general, the property that an operation remains enabled until it terminates is called persistence.

Either q_3, q_7 or q_8 could follow q_2, depending on the time from enabling to completion of c and d, and the outcome of d. Some theories make assumptions about timing, either deterministically or probabilistically, in order to reduce the number of states that must be considered. This theory does not; it views execution times and other delays as unpredictable.

NEW TRANSITIONS

The transitions from the new states are determined by the fact that commuting operations, by definition, have the same net effect regardless of the order in which they terminate. Thus, the transition sequence d_1c_1 from q_2 should lead to the same state as c_1d_1, namely q_4; and similarly for d_2c_1.

This brings us to the parallel flowchart in Figure 5. This one is equivalent to, but more parallel than, Figure 2; but it is not maximally parallel. We could increase the parallelism further, for example, by enabling d in q_1.

MAXIMAL PARALLELISM: CLOSURE

In general, an operation b may be enabled in a state q_i just prior to one in which it is presently enabled if

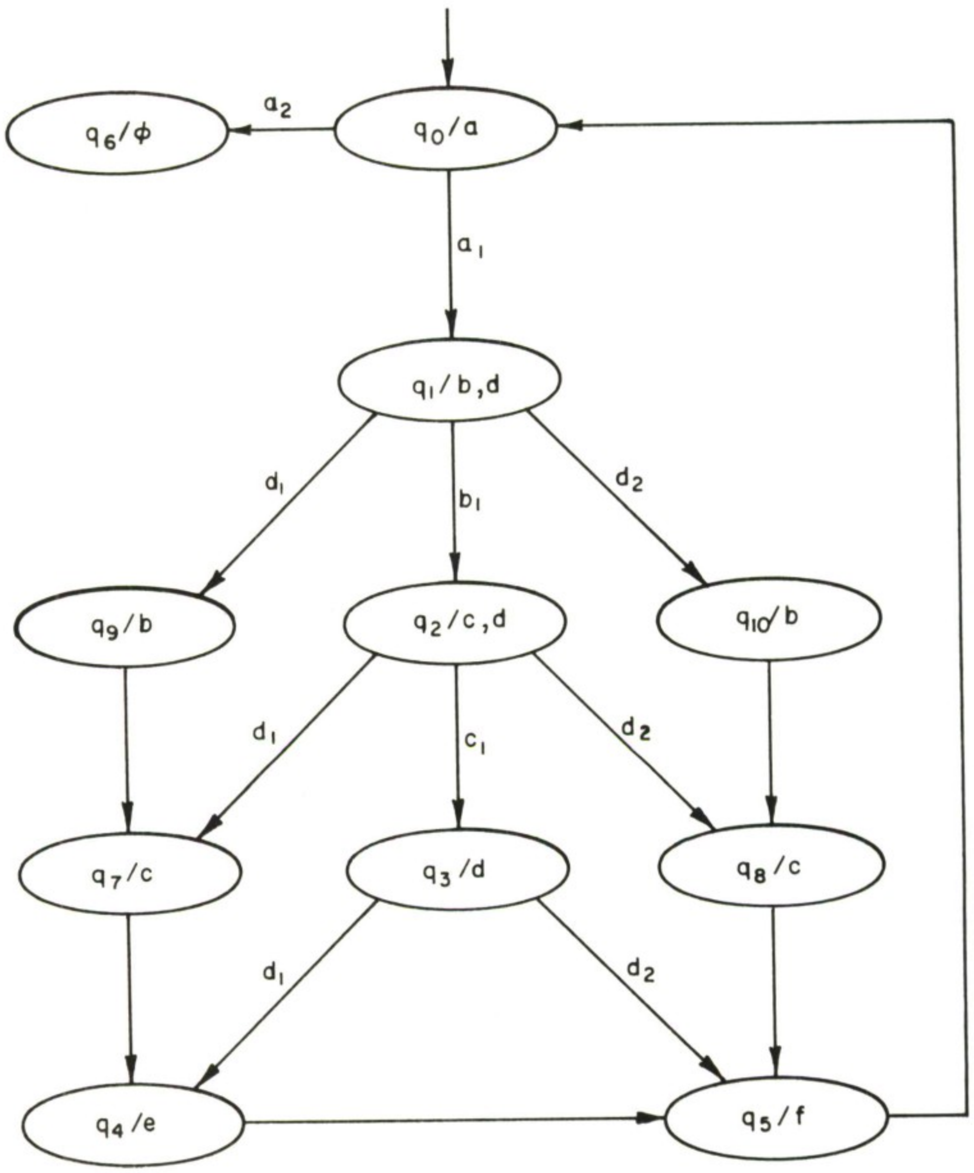
1. it commutes with the operations enabled in q_i, and
2. if all transitions from q_i go to a state in which b is enabled.

By repeatedly applying this rule to enable operations earlier when possible, and adding all new states and transitions from them each time as above, one arrives at a maximally parallel schema, called the _closure_ of the original serial one. In some cases, the closure is infinite. In this example, the closure is finite and shown in Figure 6.

The reader may verify that the schema whose flowchart is exhibited in Figure 7, has an infinite closure, assuming that its operations b and c commute. Note, however, that if there were upper and lower bounds for the time between enabling and termination for both b and c, then only a finite portion of the closure would ever be used; this observation applies to all cases of infinite closures.

DEALING WITH LARGE AND INFINITE CLOSURES

When the closure is infinite, or even just very large, there are two ways of handling the situation. One is to stop producing



1A-44,400

Figure 6 THE CLOSURE OF THE FLOWCHART IN FIGURE 2

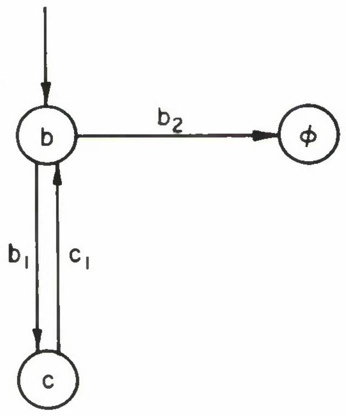
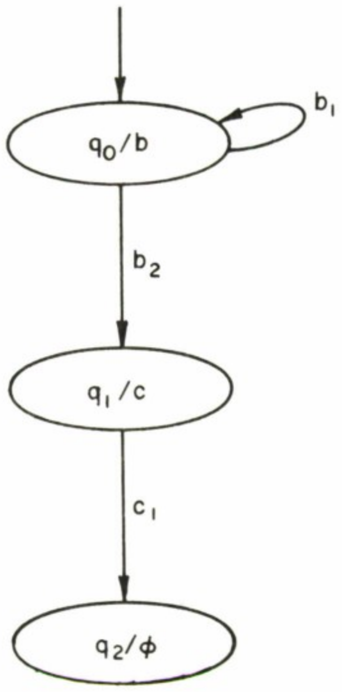


Figure 7 A FLOWCHART WITH AN INFINITE CLOSURE IF b AND c COMMUTE



IA-44,396

Figure 8 A FLOWCHART WITH AN INFINITE COMPUTATION

new states after a while; the decision as to when to stop would be based on:

1. the available time and storage space,
2. the enabling-to-termination time bounds, if any.

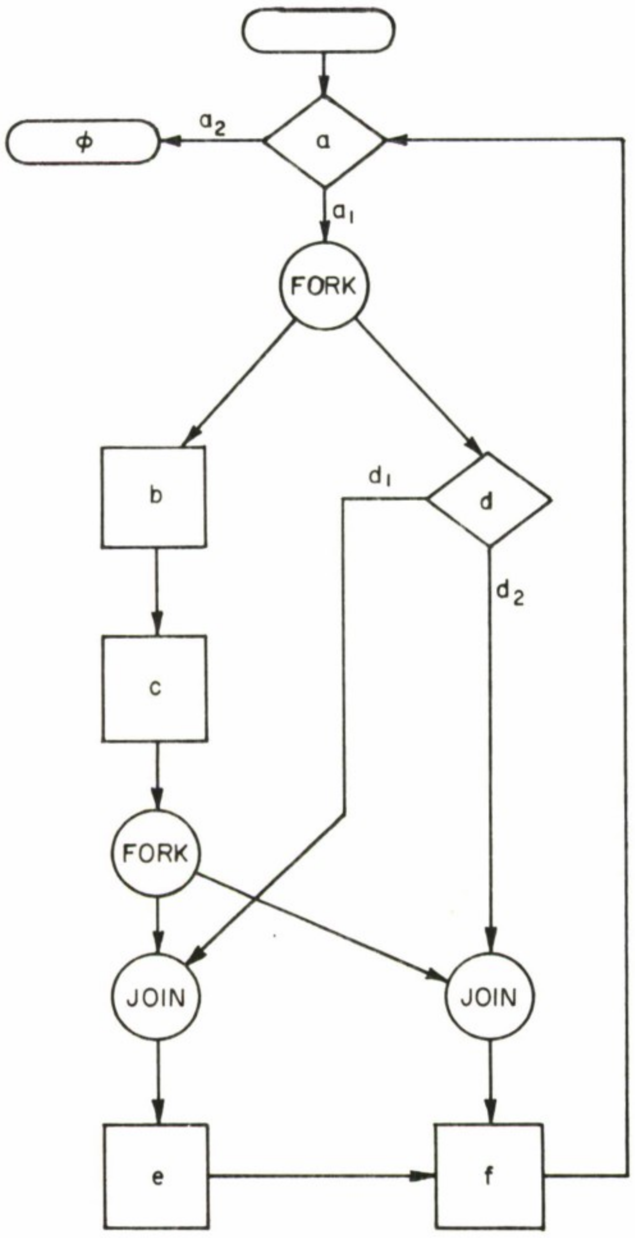
The other way is to produce new states only as needed, at execution time. It would be difficult to do this by the procedure described above, but we shall discuss better ways of doing it after the parallel derivative has been introduced.

INFINITE COMPUTATIONS

Consider the flowchart in Figure 8. It is maximally parallel even if we assume that b commutes with c. We are not permitted to enable c in q_0 because the b_1 transition from q_0 goes to a state (q_0 itself) in which c is not presently enabled. The reason that the rule is not modified for such cases is that operation b might terminate with the b_1 outcome forever, preventing c from ever being executed. The sequence $b_1b_1b_1$,...is an infinite computation.

Infinite computations are the source of most of the mathematical difficulties in this subject. It is tempting to exclude them from consideration for practical applications, but there are practical reasons for retaining them.

If infinite computations were disallowed, we would know that q_1 would eventually be entered and c enabled. If c does not interfere with b, there would be no formal obstacle to enabling c in q_0.



IA-44,389

Figure 9 AN ATTEMPT TO EXPRESS FIGURE 6 IN FORK-JOIN NOTATION

But suppose b were some computational loop and c were the instruction to print: "THE COMPUTATION IS DONE". If c were enabled in q_0, the message could appear before the computation was finished, which would defeat the purpose of the message even if we were sure that b would finish eventually.

One way to fix this particular problem without permitting infinite computations is to force b to signal c, which amounts to an interference that would prevent c from being enabled too early. Still, until we are confident that all such problems are understood, it is safer to follow the standard theory.

FORK-JOIN INSUFFICIENCY

The parallel flowchart notation may appear unnecessarily detailed compared with more streamlined notations such as the FORK-JOIN notation; but it is more powerful. Even the simple flowchart in Figure 6 illustrates the greater generality of the parallel program schema notation. We claim that Figure 6 cannot be represented equivalently in FORK-JOIN notation. A plausible attempt is shown in Figure 9.

Note that f is required to wait for e if e is executed, or both c and d if not. But there is a problem here. Suppose the d_1 decision is made in the first pass through the main loop, and d_2 in the second pass. The JOIN below d receives a signal from the FORK below c, but none from d_2 in the first pass. In the second pass, the JOIN below d will initiate f as soon as d is complete, without waiting for c, because a signal from c is still there, left over from the last pass. Therefore this setup does not work properly.

Although it has not been proved that no other FORK-JOIN scheme would handle this example, it seems justified to conjecture that none would. A similar example appears in [6].

SECTION III

PROGRAMS TO FLOWCHARTS

INTRODUCTION

The techniques presented in this report for finding the closure, or something close to it, begin with the serial flowchart and some abstract operations, which are specified only by the fact of their use of certain memory locations. The commutativity of operations then has to be determined by inspecting their common memory references.

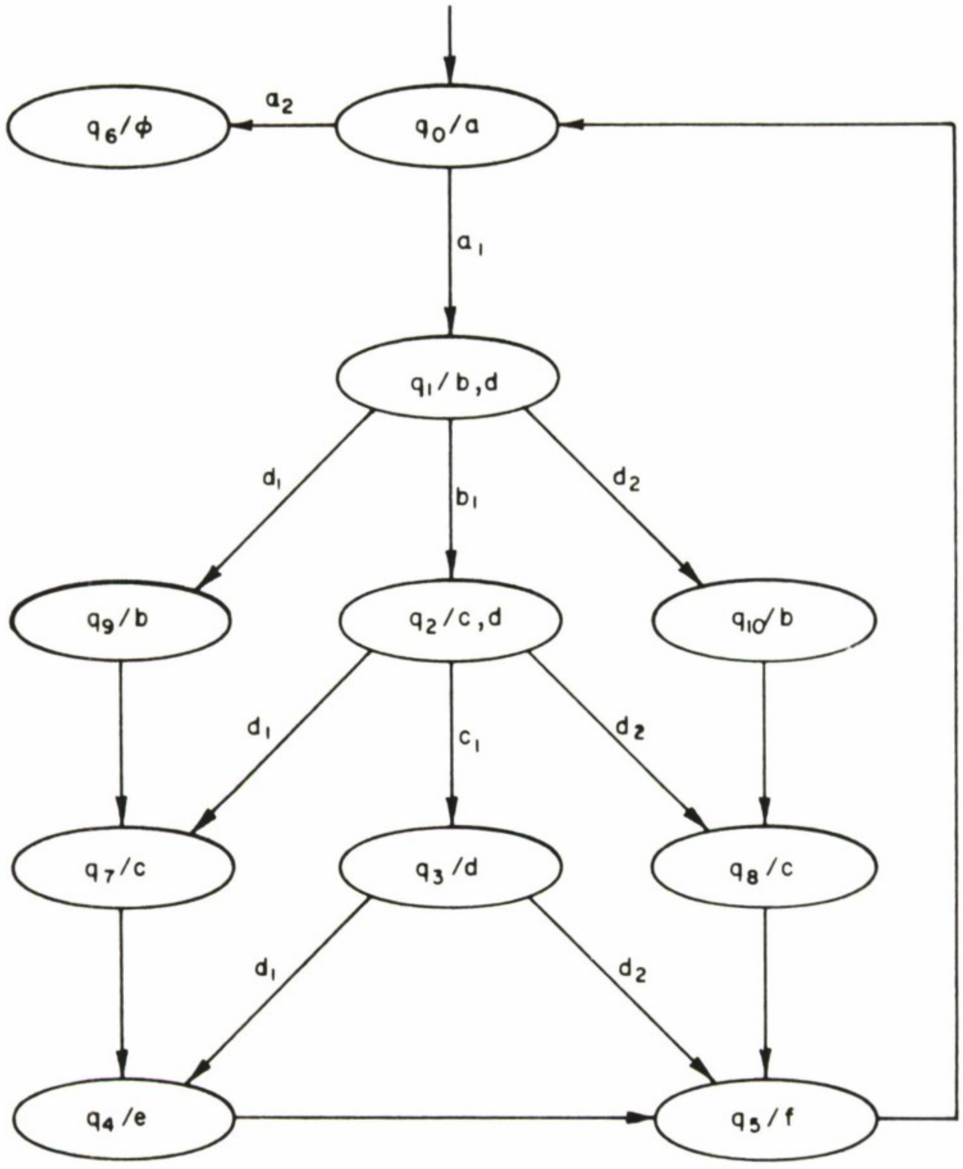
Starting with a program, its instructions must be grouped or subdivided into operations, and its variables grouped or subdivided into memory locations. This process is not as systematic at present as the process of constructing a closure from a serial flowchart. Choices have to be made without unambiguous guidance, and the decisions have far-reaching consequences.

The purpose of this section is to illustrate some of the flexibility that exists in the conversion from program to flowchart, by means of three case studies that arise in the program used as an example in Section II. They address the issues of:

(1) recombination of statements

(2) commutativity

(3) adding or subdividing variables



IA-44,401

Figure 10 THE CLOSURE OF THE EXAMPLE IN SECTION II
(REPRODUCTION OF FIGURE 6)

RECOMBINATION OF STATEMENTS

Inspection of the closure in the example in Section II, reproduced here as Figure 10, shows that operations b and c are treated alike except that c follows b. No parallelism is lost if we recombine b and c into a single operation, say b' , which consists of the two statements:

$$Z \leftarrow X$$
$$Z \leftarrow X^2$$

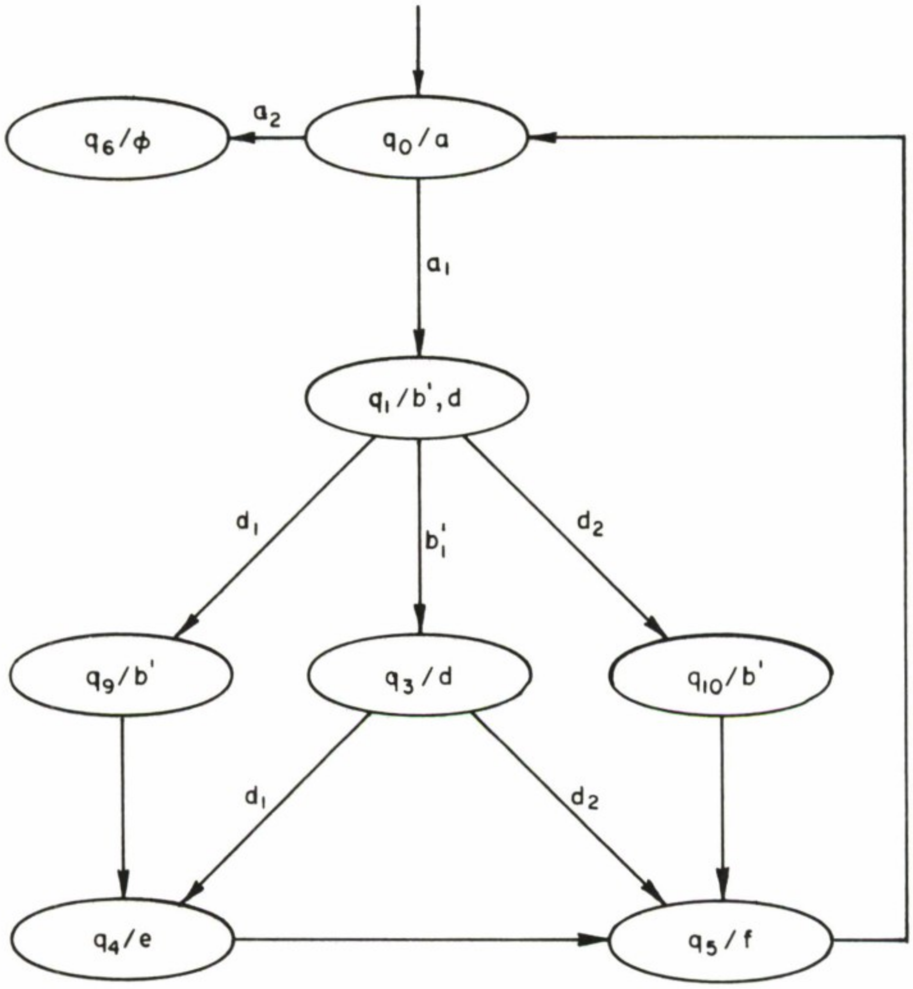
where, as before, Z represents the output file. As a result, the closure would look like the one in Figure 11. Not only is there a decrease in storage requirements, but at run time the scheduler overhead between b and c is eliminated.

COMMUTATIVITY

Consider operations c and e:

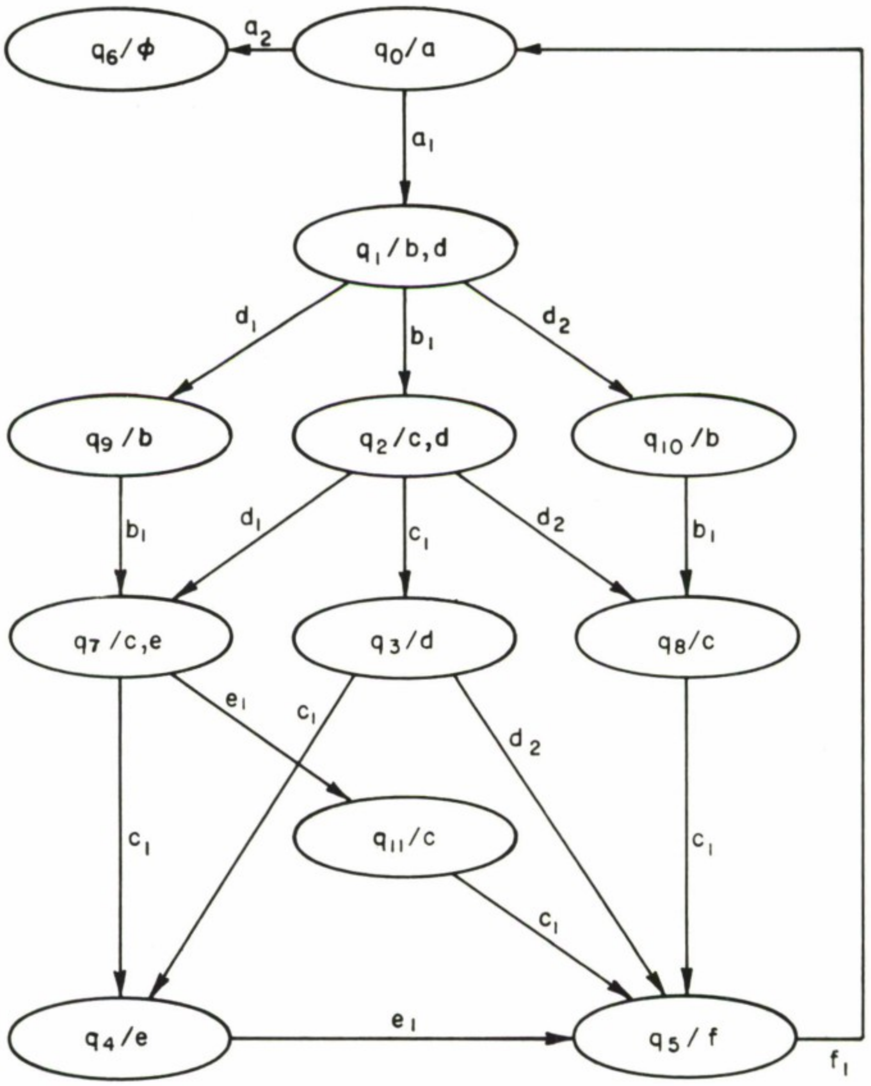
$$c\colon Z \leftarrow X^2$$
$$e\colon X \leftarrow -X.$$

On the basis of memory locations referenced we would have to say that c and e do not commute, since e writes into the variable X, which is read by c. Because of the peculiarity of the square function, however, the value of X^2 is the same whether X has been negated or not. Thus these two operations really do commute, although this fact was not detectable from the parallel program schema alone. Because they commute, the closure can be made more parallel, as in Figure 12.



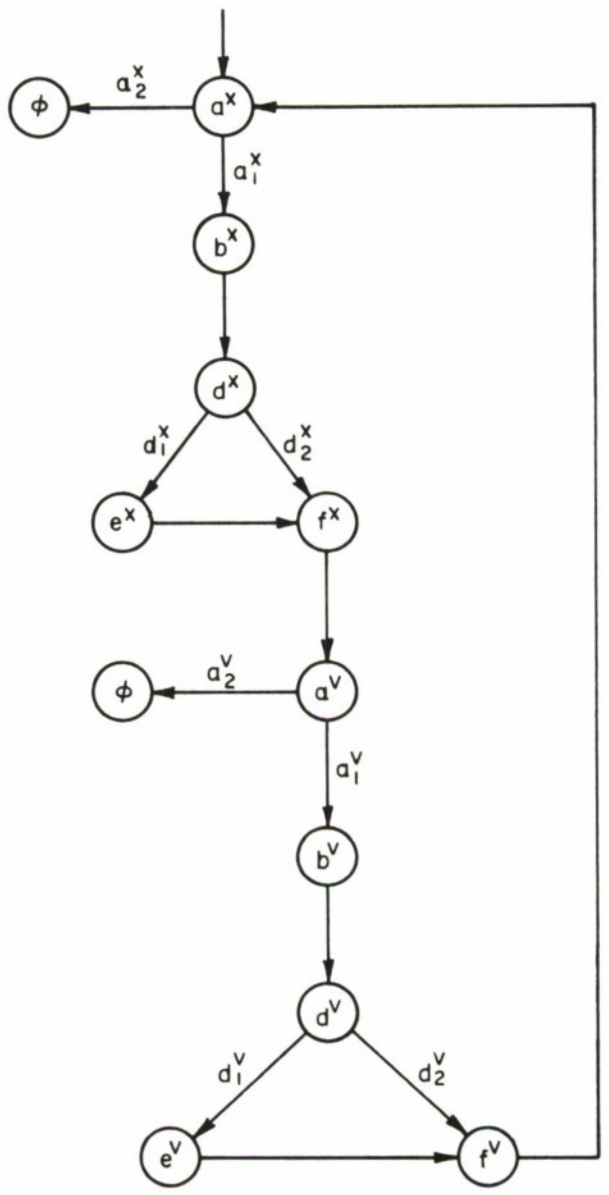
IA-44,395

Figure 11 THE NEW CLOSURE AFTER b AND c HAVE BEEN RECOMBINED INTO b'



IA-44,394

Figure 12 THE NEW CLOSURE IF c AND e ARE KNOWN TO COMMUTE



1A-44,390

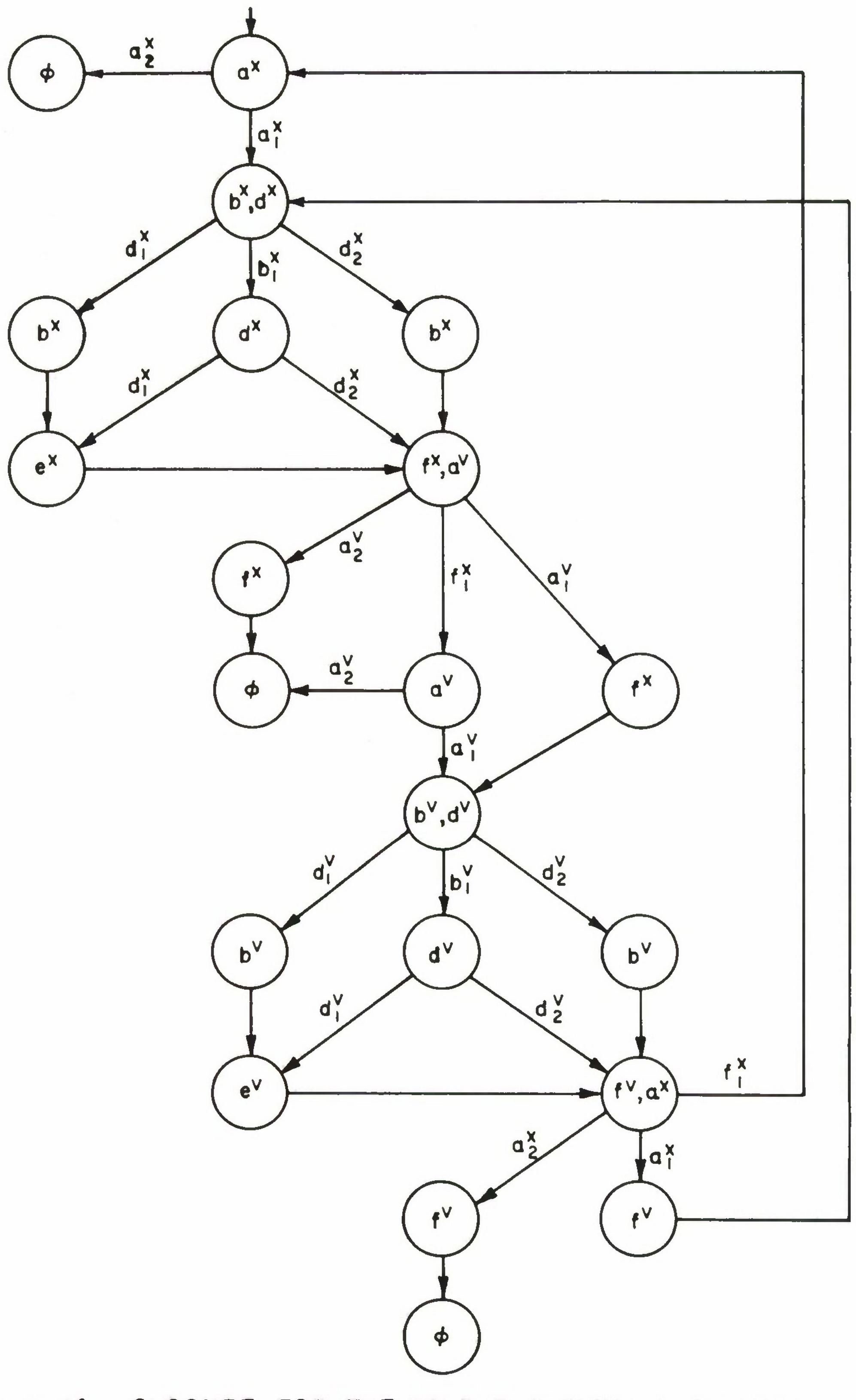
Figure 13 MODIFIED SERIAL FLOWCHART FOR DOUBLE-BUFFERING

This change in the commutativity relation and the recombination of b and c are mutually exclusive, so a choice would have to be made between them. The decision would depend on which scheduling method was being used, but recombination would usually be preferable if the operations were simple, as in this example, and the extra parallelism preferable if the operations happened to be long and time-consuming.

ADDITIONAL VARIABLES

A program like that in the example which reads input records successively can often be improved by double-buffering, in which the next input record is read while the last is being processed. To make this possible in the example we have to provide another variable besides X for input records, say V. We can now leave it up to the automatic process of constructing the closure to tell us when the read can be initiated so as to be executing in parallel with other operations.

The serial flowchart we start with is shown in Figure 13. It completes the processing of the two buffers alternately; operations on X and V are distinguished by superscripts. To simplify the analysis, we have recombined operations b and c into b. The commutativity relation is shown in Table IV, and the resulting closure in Figure 14. Note that in the closure, the processing of the two buffers is still separated; this is because the order of the output records still must be preserved. But the reading of the next



1A-44,406

Figure 14 CLOSURE FOR THE DOUBLE-BUFFERING EXAMPLE

record has been enabled immediately after the last record has been read, so that it does execute in parallel with the processing of the last record.

Table IV

Commutativity Relation for the Double-Buffering Example

OPERATION	COMMUTES WITH
a^X	b^V, c^V, d^V, e^V, f^V
b^X	$d^X, a^V, c^V, d^V, e^V, f^V$
c^X	d^X, a^V, b^V, d^V, f^V
d^X	$b^X, c^X, f^X, a^V, b^V, c^V, d^V, e^V, f^V$
e^X	$a^V, b^V, c^V, d^V, e^V, f^V$
f^X	$d^X, a^V, b^V, c^V, d^V, e^V$
a^V	b^X, c^X, d^X, e^X, f^X
b^V	$d^V, a^X, c^X, d^X, e^X, f^X$
c^V	d^V, a^X, b^X, d^X, f^X
d^V	$b^V, c^V, f^V, a^X, b^X, c^X, d^X, e^X, f^X$
e^V	$a^X, b^X, c^X, d^X, e^X, f^X$
f^V	$d^V, a^X, b^X, c^X, d^X, e^X$

There is no reason to stop at two buffers; any number could be provided, and the generation of the closure will see to it that they are used as efficiently as possible.

SECTION IV

THE PARALLEL DERIVATIVE

THE SERIAL DERIVATIVE OPERATION

Given a flowchart, a state q_j in which an operation b is enabled, and a terminator b_i of b, let

$$D_{b_i}(q_j)$$

be the state q_k so that the transition labelled b_i leads from q_j to q_k. If b is not enabled in q_j, then

$$D_{b_i}(q_j) = \emptyset.$$

The operator D should be thought of as generating the flowchart, even though in practice it is defined by the flowchart. We will call D a <u>serial derivative operator</u>, even though it may generate a flowchart which is not serial.

The operator D can be extended to sequences of terminators in a simple way. The value of $D_{c_k b_i}(q_j)$, for example, should be the state arrived at by taking the c_k transition from q_j (to $D_{c_k}(q_j)$) and then the b_i transition from there. In general, if u is a sequence of terminators, we define

$$D_{ub_i}(q_j) = D_{b_i} D_u(q_j).$$

If the sequence of length zero is denoted by λ, we may define

$$D_{\lambda}(q_j) = q_j.$$

THE PARALLEL DERIVATIVE

Like the serial derivative operator, the parallel derivative operator is defined on the states of the flowchart. The difference is that the parallel derivative operator generates the closure of a given serial flowchart.

One of the properties of the parallel derivative is that it equals the serial derivative when the latter is non-$\emptyset$. In other words,

$$\text{if } D_{b_i}(q_j) \neq \emptyset, \text{ then } P_{b_i}(q_j) = D_{b_i}(q_j). \qquad (*)$$

This is natural because the closure includes the states and transitions of the original flowchart. Since it also enables additional operations in some states, there are states q_j and operations b such that $D_{b_i}(q_j) = \emptyset$, but $P_{b_i}(q_j) \neq \emptyset$. The value of $P_{b_i}(q_j)$ in this case is an expression representing a new state.

We will say that "b is D-enabled in q_j" if $D_{b_1}(q_j) \neq \emptyset$ and "b is P-enabled in q_j" if $P_{b_1}(q_j) \neq \emptyset$.

If b is P-enabled but not D-enabled in q_j, then $P_{b_i}(q_j)$ satisfies the recursive formula:

$$P_{b_i}(q_j) = \sum_{d_k} d_k P_{b_i} D_{d_k}(q_j) \qquad (**)$$

where d is D-enabled in q_j. This formula holds only if:

(1) the flowchart defined by D is serial

(2) b commutes with d

(3) each term in the sum is non-$\emptyset$.

Condition (3) is important because it means that each parallel derivative on the right side of (**) must be expanded to find out whether it is $\emptyset$; and if any term is $\emptyset$, then the whole parallel derivative is $\emptyset$. The expansion process is essentially a search down all paths from q_j looking for states in which b is enabled; condition (3) says that b must be enabled eventually in every path, or else the parallel derivative is $\emptyset$.

The new states produced by formula (**) also have serial and parallel derivatives. Since the parallel derivative is defined in terms of the serial derivative, it suffices to give a formula for finding the serial derivative of an expression in the form of the right side of (**).

$$D_{d_m}\left[\sum_{d_k} d_k P_{b_i} D_{d_k}(q_j)\right] = P_{b_i} D_{d_m}(q_j). \qquad (***)$$

```
P($b_i$,$q_j$,PENDING)
BEGIN
    IF $q_j$ is final
    THEN RETURN($\emptyset$)
    LET d be the operator so that $D_{d_1}$ ($q_j$) $\neq \emptyset$
    IF b = d
    THEN RETURN($D_{b_i}$ ($q_j$))
    IF b and d do not commute
    THEN RETURN($\emptyset$)
    TERMS = $\emptyset$
    DO FOR ALL terminators $d_k$ of d:
    BEGIN
        X = $D_{d_k}$ ($q_j$)
        IF X = $q_m$ such that $m \leq n$
        THEN
            IF ($b_i$,$q_m$) is in PENDING
            THEN RETURN($\emptyset$)
            ELSE Y = P($b_i$,X,PENDING + ($b_i$,$q_m$))
            IF Y = $\emptyset$
            THEN RETURN($\emptyset$)
            ELSE TERMS = TERMS + $d_k$Y
    END
    RETURN(TERMS)
END
```

Program A. Recursive Procedure for Finding a Parallel Derivative

Thus the serial derivative with respect to d_m is just the d_m term of the sum, trimmed of its initial d_m. The serial derivative is $\emptyset$ with respect to terminators of operations other than d.

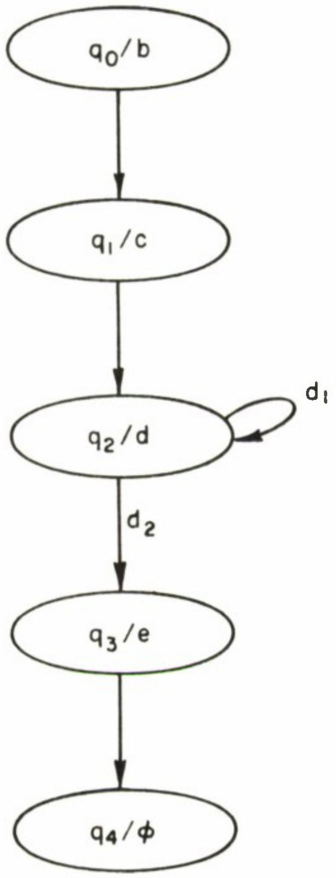
A RECURSIVE PROCEDURE FOR CALCULATING PARALLEL DERIVATIVES

An algorithm for performing the expansion in (**) is given as Program A. It is a recursive procedure with three parameters: b_i, a terminator; q_j, a state or new state expression; and an auxiliary parameter PENDING which is a list of intermediate parallel derivatives in the process of expansion. The procedure would be started with PENDING empty. It is not written in any particular language, but it should be readable to anyone knowing any high-level language which permits recursive procedures.

We assume that the states in the original flowchart are $q_o, \ldots, q_n$, and that new state expressions are named with q_i so that $i > n$. Only parallel derivatives of q_i for $i \leq n$ are put on the PENDING list.

In practice, once a parallel derivative of one of the original states is found, it may be placed on a global DONE list which will be checked by the procedure before it calls itself recursively.

The algorithm contains a bit more information than formulas (*) and (**), because of its use of the PENDING parameter. If there were no such parameter, the algorithm could become caught in an infinite loop. Instead, when an argument pair on the PENDING list is revisited, the algorithm sets the whole parallel derivative to $\emptyset$.



IA-44,393

Figure 15 FLOWCHART EXAMPLE FOR GENERATING THE CLOSURE WITH THE PARALLEL DERIVATIVE

This is not the only reasonable action that could be taken; taking it is a design choice for the definition of the parallel derivative. The parallel derivative has a non-algorithmic mathematical definition which is beyond the scope of this report, but it is given in M74-205[6], which also contains the proof that the parallel derivative generates the closure, and the proof of formulas (*) and (**).

EXAMPLES OF GENERATING THE CLOSURE

Consider the flowchart in Figure 15. Assume that the commutativity relation among the operations is given by Table V. Our procedure is to go through the states in numerical order, enabling all possible operations, and adding the transitions for their outcomes.

Table V

Commutativity Relation for the Example

OPERATION	COMMUTES WITH
b	e
c	d
d	c
e	b

We begin with q_0. We already know that the original flowchart is included in the closure, so we do not have to bother with operation b or its transition. No other operation is D-enabled in q_0, so if any are P-enabled it must be by virtue of formula (**). This immediately eliminates operations which do not commute with b. This leaves e. Applying formula (**),

$$P_{e_1}(q_0) = b_1 P_{e_1} D_{b_1}(q_0)$$

$$= b_1 P_{e_1}(q_1).$$

But $P_{e_1}(q_1) = \emptyset$ because c is D-enabled in q_1 and e does not commute with c. Thus only b is P-enabled in q_0.

Now try q_1, in which c is D-enabled. Only d commutes with c. By (**),

$$P_{d_1}(q_1) = c_1 P_{d_1} D_{c_1}(q_1)$$

$$= c_1 P_{d_1}(q_2)$$

$$= c_1 q_2.$$

This expression represents a new state; let us call it q_5 and record that

$$q_5 = c_1 q_2.$$

Since $P_{d_1}(q_1) \neq \emptyset$, we know that d is P-enabled in q_1, and we have already found the d_1 transition. We must still find the d_2 transition.

$$P_{d_2}(q_1) = c_1 P_{d_2}(q_2)$$

$$= c_1 q_3.$$

This is another new state; call it q_6 and record that

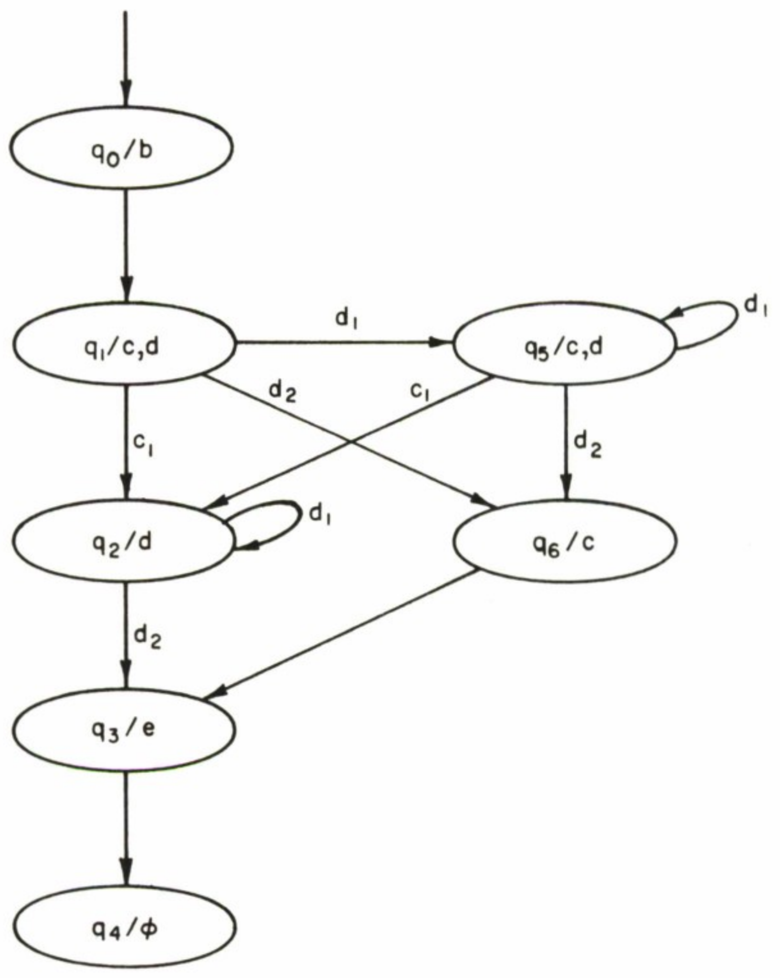
$$q_6 = c_1 q_3.$$

This completes the transitions from q_1.

Now try q_2, in which d is D-enabled. Operation c commutes with d. But

$$P_{c_1}(q_2) = d_1 P_{c_1} D_{d_1}(q_2) + d_2 P_{c_1} D_{d_2}(q_2)$$

$$= d_1 P_{c_1}(q_2) + d_2 P_{c_1}(q_3).$$



IA-44,392

Figure 16 CLOSURE FOR THE EXAMPLE

Note that $P_{c_1}(q_2)$ has appeared on the right side of the equation. This is a loop situation in which the algorithm returns immediately with $\emptyset$ as its value. Thus only d is P-enabled in q_2.

Now try q_3, in which e is D-enabled. Operation b commutes with e.

$$P_{b_1}(q_3) = e_1 P_{b_1} D_{e_1}(q_3)$$

$$= e_1 P_{b_1}(q_4).$$

But no operations are D-enabled in q_4, and therefore no operations are P-enabled in q_4 either. Thus b is not P-enabled in q_3, so only e is P-enabled there.

Since nothing is P-enabled in q_4, let us continue to the first new state, q_5. Note that c is D-enabled in q_5, since

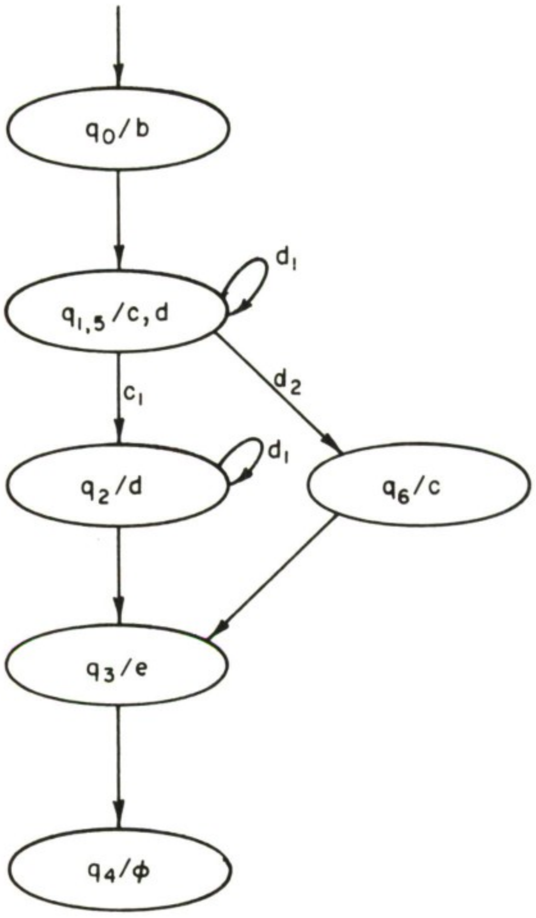
$$D_{c_1}(q_5) = D_{c_1}(c_1 q_2) = q_2 \quad \text{by (***)}.$$

Testing to see whether d may be P-enabled in q_5,

$$P_{d_1}(q_5) = c_1 P_{d_1} D_{c_1}(q_5)$$

$$= c_1 P_{d_1}(q_2)$$

$$= c_1 q_2.$$



1A-44, 391

Figure 17 CLOSURE WITH q_1 AND q_5 MERGED INTO $q_{1,5}$

This is q_5 again. Furthermore,

$$P_{d_2}(q_5) = c_1 P_{d_2}(q_2)$$

$$= c_1 q_3 = q_6.$$

Thus all transitions from q_5 lead to existing states.

Now try q_6. Operation c is D-enabled in q_6 and its c_1 transition leads to q_3. Only d commutes with c; but

$$P_{d_1}(q_6) = c_1 P_{d_1}(q_3) = \emptyset \ .$$

Therefore only c is enabled in q_6. Since q_6 is the last new state, the closure has been completed; it is shown in Figure 16.

Note that q_1 and q_5 could be merged into a single state, as in Figure 17. State q_5 was really not a new state. Any state is completely determined by its serial derivatives, and q_5 has the same serial derivatives as q_1. This might have been detected at the time q_5 was produced if q_1 and the other original states had been given expressions which show their serial derivatives. In the case of q_1, we could have written:

$$q_1 = c_1 q_2.$$

Then, when the expression c_1q_2 occurred as a parallel derivative, it would have been recognized as q_1 instead of being called a new state.

SECTION V

THE QUEUE REALIZATION

INTRODUCTION

Given a serial flowchart, a queue realization can be constructed from it which represents the flowchart of a more parallel schema, often its closure. The queue realization is due to Keller [5]. A queue realization has two parts:

(1) a finite-state part

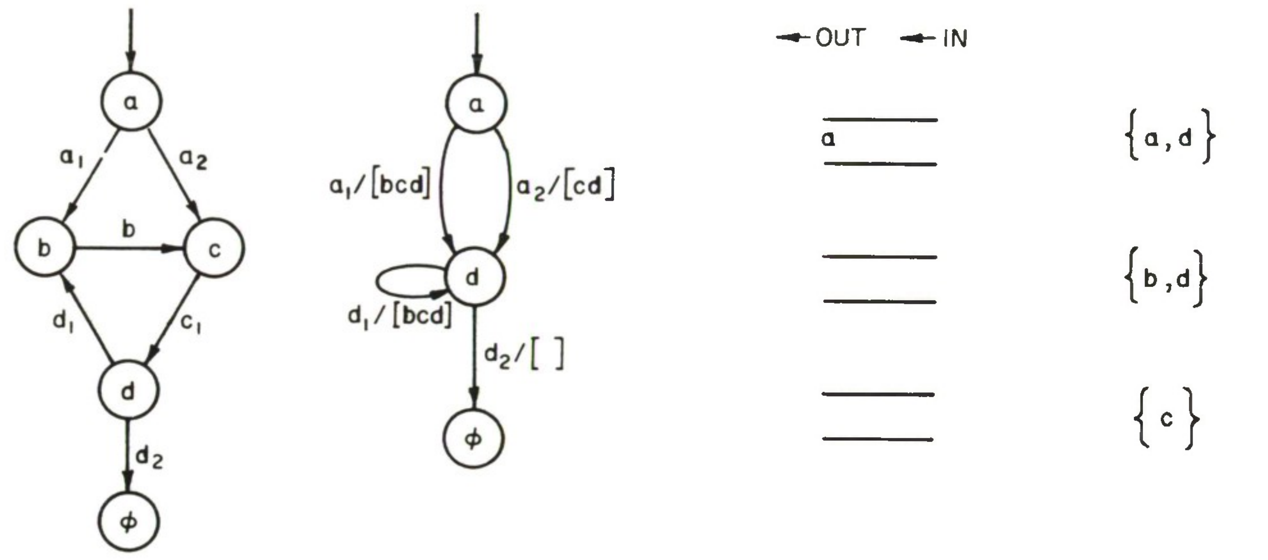
(2) a finite set of queues.

The state of the parallel flowchart is represented by the combined state of the two parts, if we regard the "state" of a set of queues to be just their contents. Since the queues are unbounded in length, the parallel flowchart may have an infinite number of states, even though it has a finite description. This makes a queue realization especially interesting as an alternative when the closure of the given serial flowchart is infinite.

FINITE-STATE PART: DECISIONS

The finite-state part is constructed as follows. Let a decision be an operation with two or more outcomes. The states of the finite-state part are those states of the serial flowchart in which a decision is enabled, plus all final states. If the initial state of the serial flowchart has a decision enabled, then it is also the initial state

IA-44,386



(a) SERIAL FLOWCHART AND FINITE-STATE PART OF QUEUE REALIZATION

(b) QUEUES WITH INPUT SETS AND INITIAL LOADING

Figure 18 FLOWCHART EXAMPLE AND QUEUE REALIZATION

of the finite-state part. Otherwise the initial state of the finite-state part is the first decision state to be reached.

Note that for each outcome of a decision, there is exactly one path that takes the flow of control to the next decision or a final state. We will put a transition from each decision state to each of the possible next decision or final states, labelled with the terminator of the appropriate outcome.

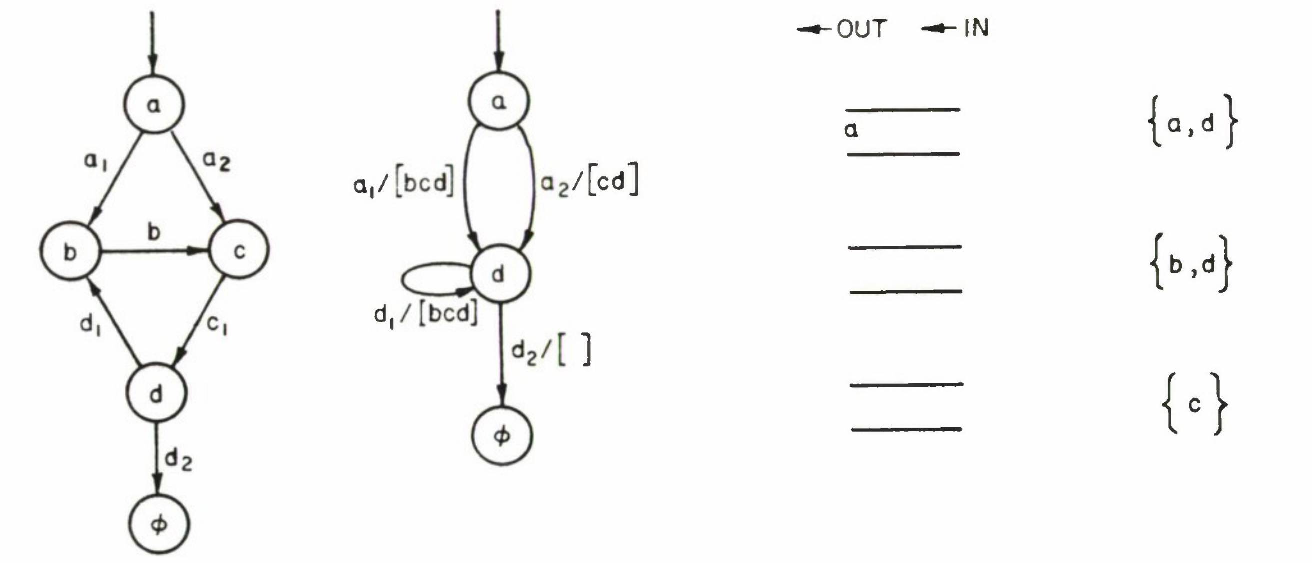
This state graph also has outputs associated with the transitions; the output is the sequence of (non-decision) operations whose terminators lead to the next decision or final state. If the path leads to a decision, that operation is added to the end of the output sequence. An example of a serial flowchart and the corresponding finite-state part of a queue realization is shown in Figure 18(a).

QUEUES

There is a queue corresponding to each pair of operations that do not commute, and an additional queue for each operation which commutes with all others. The set of operations corresponding to a queue form its <u>input set</u>; only members of its input set are ever entered into a queue.

Optionally, the number of queues can be reduced by merging input sets all of whose members are non-commuting. For example, if each pair of b, c, and d are non-commuting, then one queue with input set {b,c,d} can replace the three with input sets {b,c}, {c,d}, and {b,d}.

IA-44,386



(a) SERIAL FLOWCHART AND FINITE-STATE PART OF QUEUE REALIZATION

(b) QUEUES WITH INPUT SETS AND INITIAL LOADING

Figure 18 FLOWCHART EXAMPLE AND QUEUE REALIZATION

QUEUE LOADING

Queues are loaded during state changes with the sequences that occur as outputs in the finite state part. Into every queue is entered those operations from the sequence in its input set. The operations are entered in the same order (left to right) as they occur in the path. Note that each operation in the sequence is entered into all queues whose input sets contain it.

The queues are loaded initially with the sequence of operations which lead from the initial state of the serial flowchart up to and including the first decision.

Assuming the commutativity relation shown in Table VI, the queues with their input sets and initial loading for the example are shown in Figure 18(b).

Table VI

Commutativity Relation for Queue Realization Example

OPERATION	COMMUTES WITH
a	b,c
b	a,c
c	a,b,d
d	c

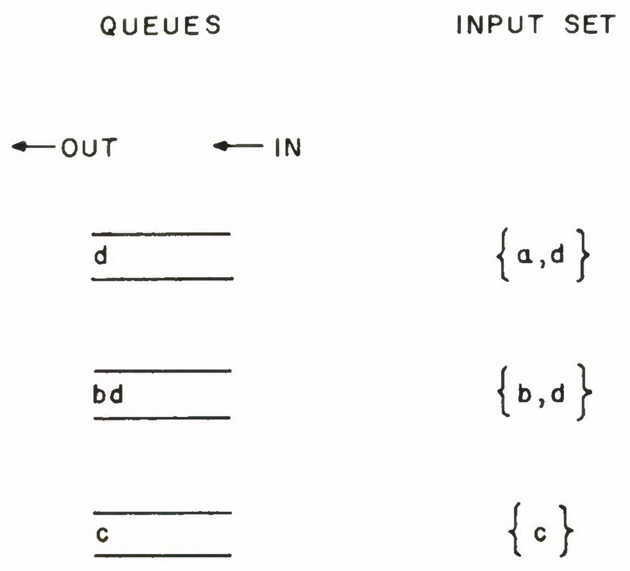


Figure 19 CONDITION OF THE QUEUES AFTER a HAS TERMINATED WITH OUTCOME a_1

ENABLED OPERATIONS

In a state of the queue realization, the operations to be considered enabled are those which appear at the front of every queue which accepts them.

TERMINATION AND TRANSITIONS

When an enabled operation has been executed and terminates, it should be removed from the front of all the queues in which it appears. This action may enable additional operations. If the operation just terminated was not a decision, the change in the queues is the only state change that takes place. If it was a decision, its outcome causes a transition in the finite state part. The output sequence for the transition is loaded at that time into the queues.

Figure 19 shows the condition of the queues after operation a in the example has terminated with outcome a_1 and the queues have been loaded with bcd. The finite-state part is now in state d, and operations b and c are enabled.

INITIATION

Operations may be initiated at any time after they become enabled. Although they remain at the fronts of the queues until they terminate, a scheduler would mark those which have been initiated to prevent other processors from re-initiating them.

This page is intentionally left blank.

PULLING TRANSFORMATION

In the queue realization described above, the finite-state part was constructed from the given serial flowchart. Keller has shown that more parallelism can often be achieved by using a different serial flowchart that is equivalent to the given one. We will give a technique for constructing such a serial flowchart using the parallel derivative.

The construction is like the generation of the closure, in that the parallel derivative is used to determine operations which are P-enabled but not D-enabled. Unlike the closure, however, only one operation is chosen to be enabled in each state. Let us call an operation Q-enabled if it is chosen to be enabled in the serial flowchart generated for the best queue realization.

Begin with the initial state. In each state, choose one operation b to be Q-enabled, such that:

(1) b is D-enabled and it is not a decision,

or (2) a decision is D-enabled, b is P-enabled, and b is not a decision,

or (3) b is D-enabled, b is a decision, but no non-decisions are P-enabled.

IA-44,385

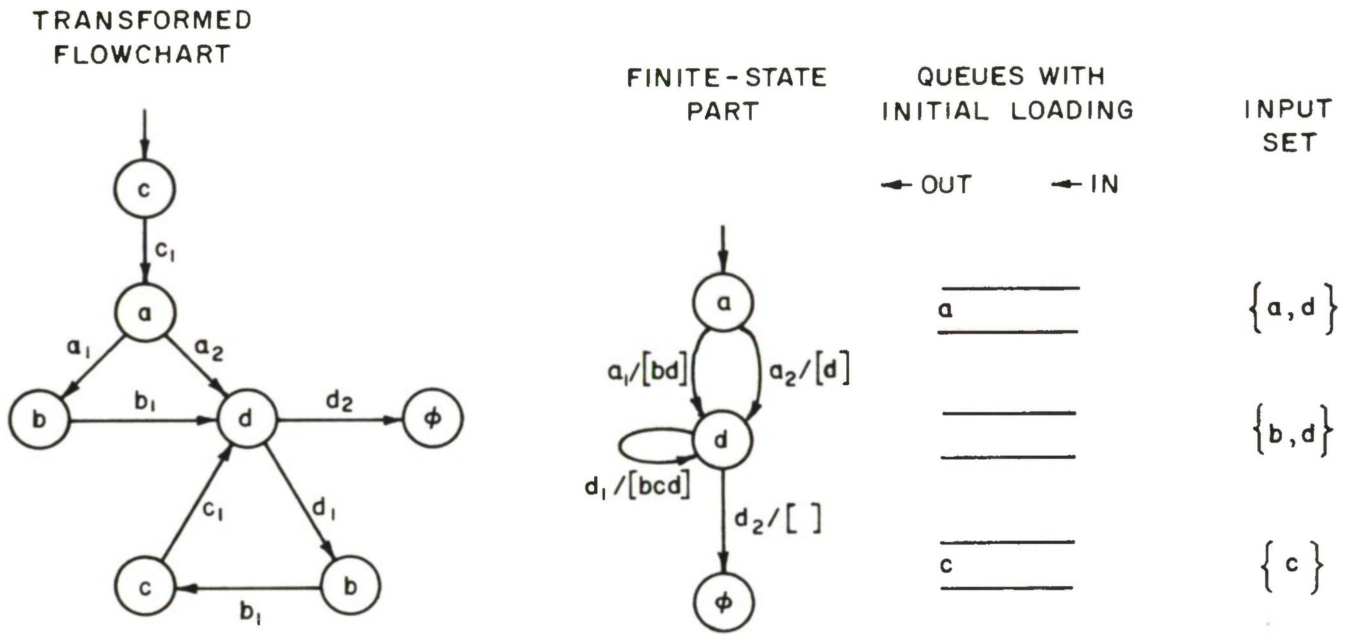


Figure 20 TRANSFORMED FLOWCHART AND ITS QUEUE REALIZATION

In short, enable a non-decision whenever possible.

Condition (2) leaves some freedom of choice, since several non-decisions might be P-enabled; the choice among them is arbitrary.

Having determined which operation is to be Q-enabled, the transitions labelled with its terminators are determined with the parallel derivative as before.

The new serial flowchart may be regarded as arising from the old one through a transformation, called the _pulling_ transformation; its net effect is to enable non-decisions as early as possible with respect to decisions. The given serial flowchart might already satisfy this condition, in which case the transformation will leave it as is.

Figure 20 shows a flowchart transformed from the flowchart in Figure 17, the finite state part, and the queues with their new initial loading. Note that additional parallelism is already evident, since two operations instead of one are enabled initially.

QUEUE REALIZATION OF THE CLOSURE: RESOLVABILITY

As remarked earlier, the queue realization represents or simulates a parallel flowchart; but this parallel flowchart is not necessarily the closure, even when a best equivalent serial flowchart is used.

It is easy to see where the queue realization may fall short. Observe that at any time there is only one decision which appears in the queues, since loading only occurs when a decision terminates. Consequently the queue realization cannot be the closure if two or more decisions are enabled in any state of the closure. Conversely, it can be shown that if no state of the closure enables two or more decisions then the queue realization is the closure, if its finite part is constructed from a transformed serial flowchart.

A serial flowchart whose closure does not enable two or more decisions in any state is called <u>resolvable</u>. It can be shown that the test for resolvability can be made just on the states of the original serial flowchart; thus the test can be made conveniently with the parallel derivative without generating the whole closure.

SECTION VI

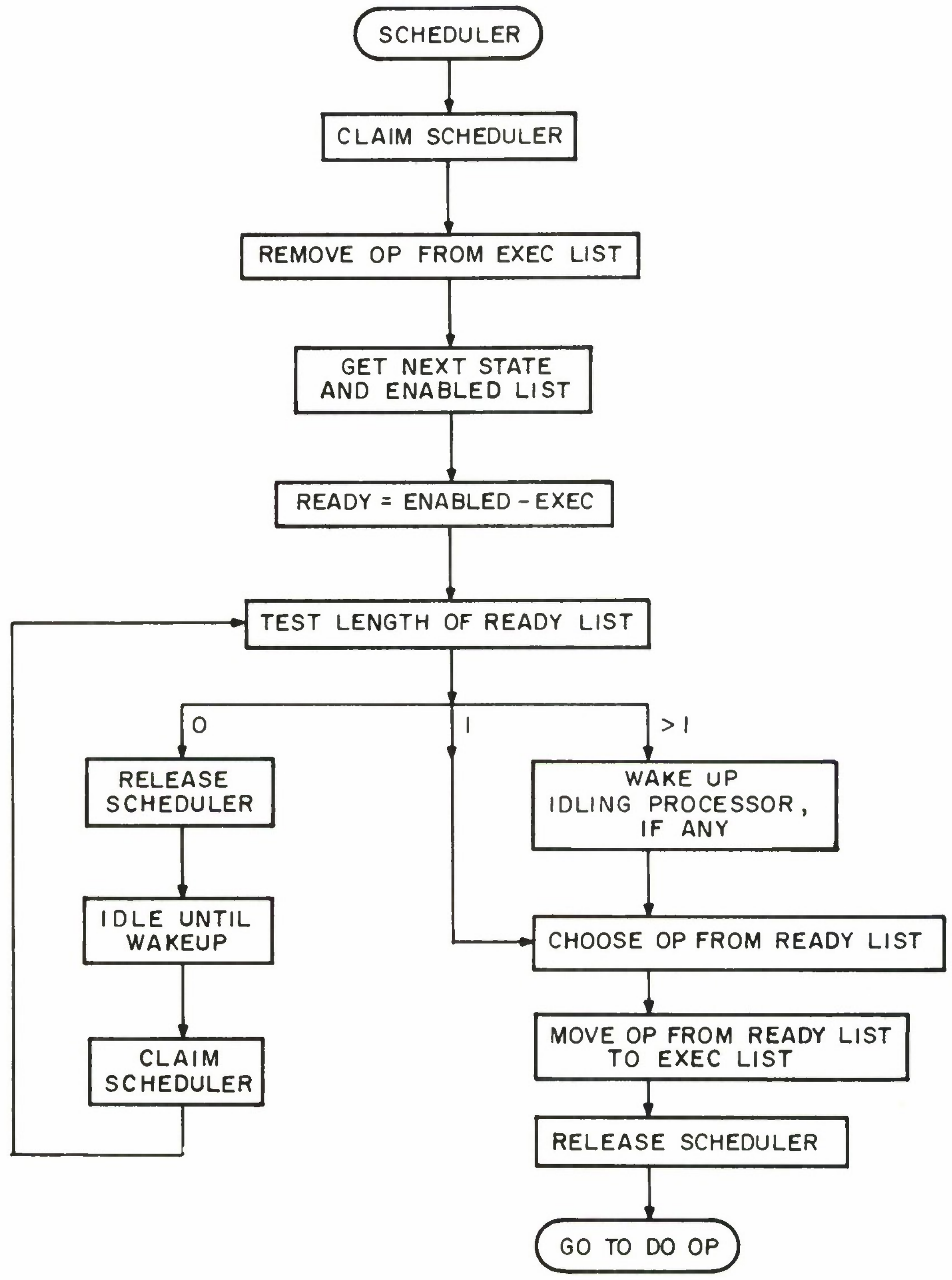
SCHEDULING

ORGANIZATION OF A SCHEDULER

A scheduler based on parallel program schemata would look like Figure 21. It would have a READY list and EXEC list of operations, which together make up the set of operations enabled in a given state. When a processor completes an operation it enters the scheduler and uses the terminator from the outcome of the operation to determine the next state, and consequently the new list of enabled operations. Subtracting from this the operations which are currently being executed, the remainder constitute the new READY list, from which the processor chooses its next operation. If the new READY list is empty, the processor leaves the scheduler and idles until it is awakened by a processor which has caused more operations to be added to the READY list.

PROTECTION

The scheduler may be executed by only one processor at a time. Otherwise the actions by two or more processors on the READY and EXEC lists could interfere, causing improper scheduling, or garbling the scheduler lists. An example of what might happen is shown in Table VII. The instructions executed by two processors are traced as they execute the scheduler, at a slight offset, and the effects on the READY, ENABLED and EXEC lists and the current state are shown. In



IA-44,404

Figure 21 SCHEDULER BASED ON PARALLEL PROGRAM SCHEMATA

this example both processors choose d as the next operation to execute, although it should only be executed by one of them.

Table VII

Possible Sequence of Events If Scheduler Is Unprotected

P_1	P_2	READY	EXEC	STATE/ ENABLE
(Executing b)	(Executing c)	∅	b,c	q_0/b,c
b Terminates				
Remove b From EXEC			c	
Get Next State	c Terminates			q_1/c,d
	Remove c From EXEC		∅	
READY←ENABLED-EXEC		c,d		
	Get Next State			q_2/d
	READY←ENABLED-EXEC	d		
Choose Operation: d				
	Choose Operation: d			

The CLAIM and RELEASE instructions in Figure 21 are intended to represent whatever instructions a processor has to go through to gain and relinquish sole access to the scheduler, including the waiting time which may be necessary in CLAIM. The actual instructions involved depend on the hardware. Some suggestions about how it might be done with the Intel equipment to be used in this project are given in Section VII.

It might seem a bit heavy-handed to protect the scheduler this way from concurrent execution. Why not analyze the scheduler to determine which of its operations could be executed in parallel? The problem is that the operations within the scheduler then have to be scheduled.

FINDING THE NEXT STATE

Determining the next state and the operations enabled in it is likely to be the most time-consuming operation of the scheduler. There are three ways of doing it:

table:	read them from a previously generated parallel flowchart.
production:	produce them by applying the parallel derivative operator.
queue:	use a queue realization.

These three ways will be compared below on the basis of:

(1) amount of parallelism

(2) preprocessing requirements

(3) run-time scheduling overhead

(4) run-time storage requirements.

AMOUNT OF PARALLELISM

Since the closure is maximally parallel, any of the methods is as good as it can be when it generates the closure. Circumstances in which a method might fail to generate the closure are the following:

(1) If the closure is infinite, or even just very large, the table method will not generate a complete closure in the available time and space.

(2) If the flowchart is not resolvable, the queue method will not generate the closure.

In case (1), the table method can still be used until the available time and space are nearly used up; then the remainder of the flowchart can be generated by enabling only D-enabled operations in each new state. The resulting flowchart will be finite, but it will sacrifice some parallelism in the states generated after the parallel derivative is put aside.

One subtlety in using the table method in such circumstances is the choice of the order in which to generate the transitions from new states. One should first complete the states which are more likely to be entered, so that they will be less likely to suffer from a reversion to the serial derivative.

In case (2), even though the queue realization is not the closure, it is not far from being maximally parallel, since the only difference is that some pair or group of decisions will be executed sequentially when they could have been done in parallel.

PREPROCESSING REQUIREMENTS

The table method requires the most preprocessing and the production method the least. All the production method needs is:

(1) the serial flowchart,

and (2) some representation of the commutativity relation between operations.

The queue method needs:

(1,2) all that the production method needs,

and (3) an application of the pulling transformation to generate a new serial flowchart, and its conversion to a finite-state part.

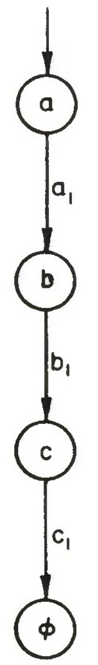
The table method also needs the serial flowchart and the commutativity relation, and in generating the closure it generates not only all the states generated by the pulling transformation, but usually a great many more.

SCHEDULING OVERHEAD

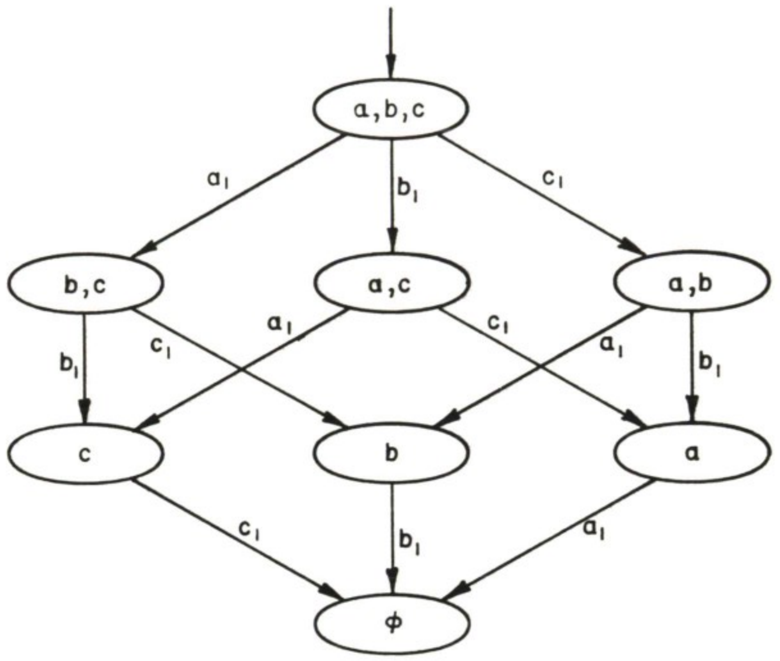
As one might expect, the more time spent preprocessing the flowchart, the less is needed at run time. The table method is by far the quickest at run time, since the new state and enabled operations are determined essentially by a table lookup. The queue realization is slower at run time because it must determine whether each of the operations brought to the front of a queue by a termination are now in front of _all_ the queues in which they occur. The time required for this test will depend on the number of queues, which in turn depends on the complexity of the non-commutativity relation.

Producing new states at run time in the production method has the drawback that each available processor must look for an operation enabled in that state. The first one can take the D-enabled operation, which is easy to find from the state expression; but other processors wishing to execute in parallel must look for some other P-enabled operation. The only way to do that is to take the parallel derivative with respect to some terminator of each of the operations that commute with the D-enabled operation until it turns out non-$\emptyset$. Since each application of the parallel derivative operator involves a tree search of some part of the serial flowchart, this process could consume a great deal of overhead time.

IA-44,403



(a) A SERIAL FLOWCHART



(b) ITS CLOSURE IF a,b,c COMMUTE

Figure 22 EXPONENTIAL BLOW-UP OF THE CLOSURE

STORAGE REQUIREMENTS

The closure of a flowchart is usually much larger than the original serial flowchart. The source of the increase in size is illustrated in Figure 22. Figure 22(a) is a serial flowchart; Figure 22(b) is its closure if all three operations commute. In situations like this a serial flowchart of size n+1 has a closure of size 2^n.

This effect could conceivably be mitigated by a less redundant representation format, such as one which took advantage of FORK and JOIN notation whenever possible. Nevertheless, the table method probably requires the most run-time storage in any event.

The queue method needs a finite-state part and queues present at run-time. The finite-state part, including outputs, is roughly the size of the serial flowchart. The queues are variable in length, and would probably share a common area, with storage allocated among the queues by a free storage list mechanism. A good size for the area would be on the order of the size of the serial program; better strategies could be worked out.

If the queue area overflows, incidentally, the scheduler could simply hold up the next decision until some of the non-decisions terminate and are removed from the queues without causing additional loadings. The minimum size for the queue area is the maximum needed for any one loading.

The production method must have on hand:

(1) the serial flowchart

(2) the current state expression

(3) workspace for calculating the parallel derivative.

The current state expression will usually be much smaller than the serial flowchart, although it could grow without bound, like the queues in the queue method, if the closure were infinite. The workspace does not have to be significantly larger than the state expression computed within it. Thus the production method needs about the same amount of space as the queue method, some small multiple of the size of the serial flowchart.

CONCLUSION

All three methods involve a significant amount of preprocessing, although the table method requires by far the most. The other two do about as much as a compiler. The production and queue methods require more time and less space for scheduling than the table method; recall that the first two need two or three times the space for a serial flowchart, while the state table for the closure would be exponentially larger. If the closure were compressed, somehow, the time required to extract the necessary information from it would increase.

The trade-off between the production and queue methods is more delicate. The fundamental advantage of the production method is its ability to generate any transition in the closure, although, in practice, time and space limitations may keep it from doing so.

All three methods deserve to be implemented: the queue method first, then the table method, then the production method, in order of ease of implementation.

This page is intentionally left blank.

SECTION VII

INTERPROCESSOR COMMUNICATION

INTRODUCTION

Program data and scheduling data are shared by some or all processors in the system. Many multiprocessing systems include a memory arbitrator for queueing access requests to a shared memory module so that they are granted one at a time at the memory cycle rate, which is usually an order of magnitude faster than the processor instruction cycle rate. No such memory arbitrator is available for the processors considered for experimentation in this project. Instead, each memory module is connected directly to a processor which is considered to own it, and any requests by other processors to access a memory module can be granted only by suspending the normal operation of the owning processor.

A method is outlined in this section for using the interrupt mechanism to permit processors to access the memory modules owned by other processors. The basic idea is this. The requesting processor interrupts the owning processor and writes its request on an output port connected directly to an input port of the owning processor. The interrupt routine in the owning processor reads the appropriate input port and then performs the requested input or output operations. It then scans the rest of its input ports to see if any other interrupts occurred while it was processing this one.

Since I/O operations are used for all data transfers, this system operates as a network of computers. The reasons for doing it this way, and the hardware and software requirements, are given below after some background on the Intel processor modules.

PROCESSOR MODULE DESCRIPTION

The Intel imm 8-82 central processor module interfaces with memory modules and I/O modules through 8-bit parallel connections, except for a 14-bit memory address connection. It also has a number of control lines, including an interrupt request input. Up to eight input ports and twenty-four output ports may be addressed by its INPUT and OUTPUT instructions. It has one 8-bit accumulator and six other index registers.

An interrupt occurring during an instruction is responded to by taking the next instruction from an 8-bit interrupt instruction port. The two most interesting choices for the interrupt instruction are:

(1) RESTART

and (2) HALT.

The RESTART instruction stores the instruction counter on the top of a seven-register stack and branches to one of eight locations coded into the instruction. Usually a RESTART is followed some time later by a RETURN, which pops the stack and branches to the address formerly at the top. These two instructions provide a subroutine call mechanism.

The HALT instruction puts the processor into a stopped state, in which it is effectively disconnected from its memory, so that the memory can be used by another processor. The stopped state can be left only via an interrupt.

DIRECT MEMORY ACCESS

When the owning processor is stopped, a memory module is available for access by another processor in the same manner by which the processor accesses its own memory. It is tempting to try to use this feature to accomplish data sharing, but there is a problem with this approach that would have to be solved first: how to deal with interrupts occurring while the owning processor is already stopped and its memory is being accessed by another processor. If the interrupt instruction is just HALT, there is nothing to prevent the new processor from attempting direct memory access at the same time, and this is disastrous in the absence of a hardware memory arbitrator. On the other hand, a RESTART would result in access of the memory for further instructions by the owning processor, again conflicting with the first processor's direct memory access. No other interrupt instruction appears to be any help.

Consequently, direct memory access to a stopped processor is not safe unless interrupts to that processor can be disabled. Although the imm 8-82 processor does not possess an instruction for disabling interrupts to itself, very little circuitry would be required to provide that capability. A suitable device would:

(1) intercept all interrupts to the processor, and

(2) enable or disable them according to a signal from an output port of the processor.

When a processor is stopped with interrupts disabled by such a device, it will remain stopped forever unless the device will also:

(3) pass an interrupt from the processor performing direct memory access.

The identity of the processor performing direct memory access could be coded into the command to disable interrupts.

COMMUNICATION BY I/O

Processors can communicate with one another without direct memory access, by using their input and output ports. An INPUT instruction copies the byte at a selected input port into the accumulator, and an OUTPUT instruction copies the accumulator into a selected output port.

Rapid transmission of data can be accomplished with two output ports of the sending processor connected directly to two input ports of the receiving processor. One connection carries the data, and the other keeps a count of the number of bytes transmitted. Synchronism is maintained between the sender and receiver as long as the sending loop takes longer than the receiving loop. If for any reason the receiver gets behind, however, it can detect the fact by means of the count, and interrupt the sender.

An interrupt is useful not only to resynchronize transmission but also to initiate it, and to send other occasional information.

GENERATING INTERRUPTS

One processor can interrupt another if one of its output ports is connected through an OR gate with the interrupt request line of the other processor. It takes four operations to produce a pulse:

(1) load accumulator with 1

(2) OUTPUT

(3) load accumulator with 0

(4) OUTPUT.

To give the interrupt routine information with which to work, the interrupting processor should send a request code to the interrupted processor before the interrupt is complete. It can use the same output port for the interrupt pulse and request code if one bit, say the

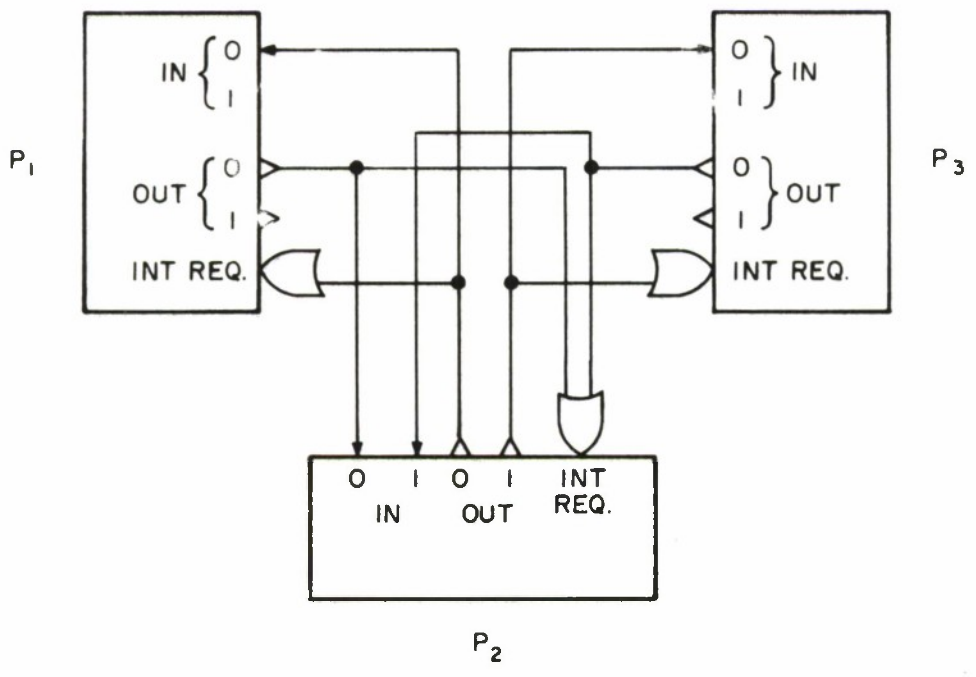


Figure 23 CONNECTIONS BETWEEN imm 8-82 PROCESSORS FOR CODED INTERRUPTS

low-order bit, of the request code is reserved for the interrupt signal. Then these four operations would be used instead:

(1) load accumulator with request code + 1

(2) OUTPUT

(3) decrement accumulator by 1

(4) OUTPUT.

The physical connections required are illustrated in Figure 23, showing three processors P_1, P_2, and P_3. For simplicity, only two input ports and two output ports are indicated. Mutual interrupt connections are shown between P_1 and P_2 and between P_2 and P_3.

INTERRUPT HANDLING ROUTINE

The program shown as Program B is a high-level representation of an interrupt handling routine that operates as described in the introduction. The interrupt instruction is assumed to be RESTART, with the beginning of this routine, INT, coded in as the branch address.

The variables mentioned are:

(1) B, an integer,

(2) I, an integer,

(3) INPUT(I), the contents of input port I,

(4) LASTIN(I), an array element in which the last value of INPUT(I) is stored.

```
INT:    B ← B + 1
        IF B ≠ 0 THEN RETURN
LOOK:   FOR I = 0 TO N DO
             IF LASTIN(I) ≠ INPUT(I) THEN GO TO PROC
        B ← B - 1
        IF B ≠ - 1 THEN GO TO LOOK
        RETURN
PROC:   <process input I>
        LASTIN(I) ← INPUT(I)
        <notify processor I>
        GO TO LOOK
```

Program B. Interrupt Routine

This routine assumes that the request code always changes from one request to the next; one bit of the request code may have to be reserved for this purpose (in addition to the interrupt bit). Thus even if two interrupts occur during the same instruction, so that only one interrupt is detected, evidence of both requests is available and shows up in the DO-loop.

The variable B indicates whether the routine

(1) is currently being executed

(2) has been interrupted.

Outside of the interrupt routine, B = -1. When the interrupt routine is entered, B is incremented. If the result is 0, the routine continues; but if the result is 1 or more then the routine must already be in execution, and the interrupt is otherwise ignored. If the routine continues, it does not return until:

(1) all inputs have been checked,

(2) the value of B indicates that no interrupts occurred part way through the checking.

The claim that interrupts occurring while the interrupt routine is in execution do not cause any failures cannot be supported from the high-level representation. The routine would first have to be coded in machine language, since only machine language instructions are indivisible by interrupts.

This page is intentionally left blank.

There is one problem that does not show up in the high-level representation that must be solved before the machine language code can be written: saving registers and flags.

SAVING REGISTERS AND FLAGS

The occurrence of an interrupt should be transparent to the main program, except, of course, for the data transfer which was the purpose of the interrupt. In particular, the contents of the accumulator and other registers used by the main program must be saved and restored by the interrupt routine if it uses them. The interrupt routine will certainly use the accumulator, and probably other registers as well.

The interrupt routine will also affect the values of the flag flip-flops tested by conditional jump instructions. The flags should be restored because of the possibility of the following sequence of events:

(1) a compare or arithmetic instruction sets the flags,

(2) the interrupt routine is executed,

(3) a conditional jump tests the flags.

An interrupt occurring when B is 0 or more is a special case because the interrupt routine returns almost immediately in this case without changing anything except B and possibly some flags. Thus only the flags need be restored, and this can be done without using any registers. We assume here that B is a register not used by the main program.

```
 INT:  If Z = 0 THEN GO TO INTA
       B ← B + 1 [IF B = 0 THEN Z ← 1 ELSE Z ← 0]
       IF Z ≠ 0 THEN GO TO RET
       <continue with interrupt routine which
        saves and restores Z = 1>

INTA:  B ← B + 1 [If B = 0 THEN Z ← 1 ELSE Z ← 0]
       IF Z ≠ 0 THEN GO TO FIX
       <continue with interrupt routine which
        saves and restores Z = 0>

 FIX:  A = 0? [IF A = 0 THEN Z ← 1 ELSE Z ← 0]
       IF Z = 0 THEN GO TO RET
       A = 1? [Z ← 0]
 RET:  RETURN
```

Program C. Flag-Saving Preamble

The flags which may be affected are saved by testing them and jumping to different copies of the routine before incrementing B. An example of how this would be done, under the simplifying assumption that only the "zero" flag (Z) is affected, is shown as Program C. Although the language in the example is a pseudo-assembly language, the instructions used correspond directly to Intel 8008 machine instructions.

The remainder of the routine does not have to be duplicated in full at places where "continue with interrupt routine" is indicated. The register-saving preamble would have to be duplicated, but then the flag information can be saved in a memory list and the rest of the routine can then be done in common.

It is worth proving explicitly that nested interrupts may occur at any time and to any depth (within the capacity of the processor stack) without derailing the saving of registers and flags or interfering in any way other than the way they are supposed to (incrementing B).

The proof is by induction on the depth of nested interrupts. If the depth is zero, that is, if the interrupt routine is not itself interrupted, there is no problem. Now assume that registers and flags are preserved if the depth of interrupts is n-1 or less; but that the n^{th} nested interrupt has just occurred. The interrupt routine is called recursively; <u>this</u> call will be interrupted at a depth one less, so by the induction hypothesis it has no net effect on registers other than B or flags. Consequently the top-level interrupted call is not disturbed, no matter which pair of instructions is separated by the nested call.

CLAIM AND RELEASE

If the interrupt routine described above is used to access the scheduler, the CLAIM and RELEASE operations are redundant. The processor which has just completed an operation simply sends the terminator to the processor owning the scheduler data, and interrupts it. When the interrupt routine processes that input, it executes the scheduler algorithm and then sends back the location of the next operation to be executed, if any.

Thus only the processor owning the scheduler data executes the scheduling algorithm. Because of the way nested interrupts are postponed, the scheduling algorithm is never reentered during execution; this makes explicit CLAIM and RELEASE operations unnecessary.

REFERENCES

1. D. J. Kuck et al, "Measurement of Parallelism in Ordinary FORTRAN Programs," Computer, January 1974, pp. 37-46.

2. M. J. Gonzalez and C. V. Ramamoorthy, "Recognition and Representation of Parallel Processable Streams in Computer Programs," Parallel Processor Systems, Technologies, and Applications (L. C. Hobbs, Ed.) Spartan Books, New York, 1970, pp. 335-374.

3. D. J. Kuck, Y. Muraoka, and S. Chen, "On the Number of Operations Simultaneously Executable in Fortran-like Programs and Their Resulting Speedup," IEEE Transactions on Computers, Vol. C-21 No. 12, December, 1972, pp. 1293-1310.

4. L. Lamport, "The Parallel Execution of DO Loops," Communications of the ACM, Vol. 17, No. 2, February 1974, pp. 83-93.

5. R. M. Keller, "Parallel Program Schemata and Maximal Parallelism," J. of the ACM, Vol. 20, No. 3, July 1973, pp. 514-537 and Vol. 20, No. 4, October 1973, pp. 696-710.

6. J. K. Millen, Construction With Parallel Derivatives of the Closure of a Parallel Program Schema, M74-205, The MITRE Corporation, December 1973.